Includes Mock Exams

Certified Kubernetes Application Developer (CKAD) Exam Success Guide

Ace your career in Kubernetes development with CKAD certification

Sethumadhavan Kishore

bpb

Certified Kubernetes Application Developer (CKAD) Exam Success Guide

Ace your career in Kubernetes development with CKAD certification

Sethumadhavan Kishore

www.bpbonline.com

Copyright © 2024 BPB Online

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor BPB Online or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

BPB Online has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BPB Online cannot guarantee the accuracy of this information.

First published: 2024

Published by BPB Online

WeWork

119 Marylebone Road

London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55515-698

www.bpbonline.com

Dedicated to

My beloved Parents:
K P Sethumadhavan
Girija Sethumadhavan
&
My wife Ambili and my little tiger Aarav

About the Author

Sethumadhavan Kishore is an esteemed IT professional with over a decade of experience in the industry. He has held various roles throughout his career, including software engineer, technical lead, and software architect. Kishore is well-versed in both Java and Python programming languages and has extensive experience working with distributed systems, Kubernetes, and OpenShift.

Kishore has been deeply involved in cutting-edge technologies throughout his career, and his expertise in Kubernetes is unparalleled. With a keen understanding of containerization and orchestration, he has successfully navigated complex deployments, leveraging Kubernetes to streamline processes, optimize resource utilization, and enhance scalability.

Kishore holds various certifications in Kubernetes. He is a Certified Kubernetes Application Developer (CKAD), Certified Kubernetes Administrator (CKA) and Certified Kubernetes Security (CKS). He scored 96% in the CKAD exam.

Kishore is an Innovator, and has successfully filed patents, published technical papers, and presented at regional technical exchanges and seminars. He has a proven track record of delivering high-quality software solutions and has a deep understanding of the intricacies of building and deploying large-scale systems.

When not immersed in technology, Kishore enjoys spending time with his family, exploring the outdoors, recording technical videos for his YouTube channel and continuously expanding his horizons by diving into new areas of interest within the IT landscape.

About the Reviewers

❖ Keshav Saxena is a seasoned DevOps engineer with over 8 years of experience in the IT industry. He has a strong background in cloud computing, having worked extensively with major cloud platforms such as AWS and GCP. His primary expertise lies in containerization and orchestration technologies, particularly Docker and Kubernetes. He has successfully implemented and managed containerized applications at scale, optimizing performance and efficiency. Keshav is proficient in infrastructure-as-code tools like Terraform and Ansible. He has leveraged these tools to automate the provisioning and configuration of cloud resources, enabling rapid deployment and easy scalability. Keshav's commitment to professional growth is evident through his certifications in Kubernetes, Terraform, and GitLab. These certifications validate his deep understanding and practical experience in these areas, making him a valuable asset to any organization. He loves reading books and watching documentaries.

❖ Pratikkumar Panchal has over eight years of professional experience. He is a skilled DevOps Cloud Engineer specializing in Kubernetes orchestration. Their expertise lies at the intersection of cloud infrastructure management and software development, enabling seamless deployment and continuous integration of complex applications. Throughout their career, Pratikkumar Panchal has demonstrated a deep understanding of cloud technologies, utilizing platforms like AWS, Azure, or Google Cloud to architect scalable and resilient solutions.

As a Kubernetes engineer, Pratikkumar Panchal has a proven track record of designing, implementing, and maintaining Kubernetes clusters, streamlining application deployment and management. Their proficiency in containerization, combined with their ability to automate processes using tools like Docker and Jenkins, has resulted in optimized workflows and increased efficiency.

Pratikkumar Panchal is dedicated to ensuring the reliability and security of systems, employing best practices in monitoring, logging, and infrastructure as code. Their strong collaboration skills have facilitated cross-functional teamwork, leading to successful project deliveries. Whether optimizing infrastructure, enhancing CI/CD pipelines, or troubleshooting complex issues, Pratikkumar Panchal continues to drive innovation and excellence in the DevOps and cloud engineering landscape.

Acknowledgements

I would like to express my deepest gratitude to the following individuals whose unwavering support and encouragement have made this book possible.

First and foremost, I am indebted to my loving family. Their patience, understanding, and belief in me have inspired me constantly. I am profoundly grateful to the dedicated team at BPB. Furthermore, I would like to acknowledge the invaluable contributions of the technical reviewer, Keshav.

I am also thankful for the support and encouragement received from my friends and colleagues. Your words of encouragement, stimulating discussions, and belief in my abilities have been instrumental in this undertaking.

Lastly, I want to extend my heartfelt appreciation to all the readers who will embark on this literary journey. Your curiosity and interest in the subject have fueled my writing passion.

In conclusion, completing this book has been a collaborative effort, and I am profoundly grateful to each and every individual who has contributed to its creation. This endeavor would not have been possible without your unwavering support, belief, and guidance.

Preface

Welcome to the Preface of this book, designed to assist you in your preparation for the Certified Kubernetes Application Developer (CKAD) exam. Whether you are an experienced developer or new to Kubernetes, this resource aims to provide you with the knowledge, skills, and confidence needed to excel in the CKAD certification process.

The CKAD exam comprehensively assesses your ability to design, develop, and deploy applications on Kubernetes clusters. It challenges your understanding of core concepts, such as creating and configuring application deployments, services, and persistent storage, as well as troubleshooting and debugging common issues within a Kubernetes environment.

Having personally navigated the CKAD certification journey, I understand the significance of a well-rounded and practical approach to exam preparation. This book results from my firsthand experience and extensive research and insights gathered from fellow CKAD-certified professionals. It has been meticulously crafted to equip you with the essential knowledge and hands-on skills necessary to ace the exam.

The chapters of this book are structured to follow the CKAD exam syllabus, offering a comprehensive and systematic approach to learning the required topics. Each chapter provides a clear explanation of the underlying concepts, accompanied by practical examples and step-by-step

instructions to ensure a solid understanding of the subject matter. In addition, you will find exercises and practice scenarios throughout the book that simulate real-world situations you may encounter in the CKAD exam.

I would like to express my deepest gratitude to the team at BPB Publications, for their support and guidance in bringing this book to fruition. Their dedication and expertise have played a crucial role in ensuring the accuracy and relevance of the content.

Lastly, I extend my heartfelt thanks to all the readers who have chosen this book as a part of their CKAD exam preparation journey. Your commitment to expanding your knowledge and skills is commendable, and I am honored to be a part of your learning experience.

Best of luck in your CKAD exam preparation, and may this book serve as a valuable resource to help you achieve your certification goals.

The book has two simple objectives:

Help you to master Kubernetes.

Help you to crack the CKAD exam.

There is currently a high demand for Kubernetes skills in the job market. According to a survey by the Cloud Native Computing Foundation (CNCF), Kubernetes is the most in-demand skill for cloud-native professionals. Many companies are adopting Kubernetes to manage their containerized applications and infrastructure, and as a result, there is a

growing need for professionals with expertise in using Kubernetes to design, deploy, and maintain cloud-native systems.

Thus, learning and mastering Kubernetes is essential.

This book is divided into 14 chapters. The details are listed below:

Chapter 1: Introduction to Kubernetes, Docker, and Minikube - In this foundation chapter, we will learn how software development and deployment have changed over the years and also do a deep dive and try to understand the world of Cloud native technologies, Container, Docker and Kubernetes. We will also learn how to set up your Minikube. Minikube is a tool that allows you to run a single-node Kubernetes cluster locally on your own machine. Minikube single node cluster will be used to execute all the practice exercises.

Chapter 2: What, Why, and How of CKAD Exam - This chapter discusses the certification, curriculum and tips and tricks needed to pass the CKAD exam. We will also cover the topic of time management and how to practice and prepare for the CKAD exam.

Chapter 3: Exploring Pod, Deployment, ReplicaSet, and Namespace - From this chapter, we will start learning Kubernetes resource objects. First, we will start learning the core concepts of Kubernetes with Pod, Deployments, ReplicaSet and namespace. We will also discuss the basic structure of Kubernetes primitives and how to interact with Kubernetes objects via the command line–based client, kubectl. At the end of the chapter, you will understand how to create Kubernetes objects

imperatively and declaratively and know how to create core Kubernetes objects, and define their most basic configuration.

Chapter 4: Deployment Strategies and Helm Packages - In this chapter, we will learn how to update apps running in a Kubernetes cluster and in an actual zero-downtime update process. We learn different deployment strategies and also learn Helm, the package manager, which is a tool to help you define, install, and upgrade applications running on Kubernetes.

Chapter 5: Pod Design and Concepts - In this chapter, you will learn many concepts related to Pod and deployment, which is essential to know to pass the exam. This chapter will teach us about labels, annotations, and deployment. We will also learn about autoscaler, how to roll out a new revision of your application, and how to roll back to a previous revision. And lastly, we'll touch on the Kubernetes primitives Job and CronJob.

Chapter 6: Multi-container Pods and Design Patterns - In this chapter, we will learn about Multi-container Pod, its need, use cases and design Patterns. We will also look into Init Container.

Chapter 7: Kubernetes Volumes and Persistence - In this chapter, we will cover Kubernetes volume types. The chapter covers, different types of volumes for reading and writing data. We will also learn how to create and configure them. Kubernetes models persist data with two primitive PV and PVC. In this chapter, we will cover Persistence volume and Persistence volume claim, how to set it, and its relationship with Pod. We will also cover the Storage class, how to set it, and its creation using PersistentVolume.

Chapter 8: Configuring Volume, Environment Variables, ConfigMap, and Secrets - In this chapter, we will learn some advanced concepts to configure the Kubernetes object. We will start first with how to configure runtime behavior using environment variables. How to configure apps through config maps and how to handle sensitive information in Kubernetes using secrets. We will also cover Volumes and how to mount them to Pod.

Chapter 9: Service Accounts Resource Quota and Security Contexts - In this chapter, we will learn how to define security context to define privilege and access control settings. We will also learn the configuration needed to assign a Service Account to a Pod and also learn about resource boundaries and how to set resource quota.

Chapter 10: Liveness and Readiness - Probes are used by Kubernetes as an indication of a pod's health and state. Health probes help automate detection and correction of issues seen in the Kubernetes cluster. These probes help in checking if the application is working as expected. In this chapter, we will discuss the various probes, namely readiness, liveness and startup probes, the difference between liveness and readiness probes and under which scenario to use. We will also learn how to configure them and how to debug scenarios associated with liveness and readiness probes.

Chapter 11: Monitoring and Debugging of Kubernetes Cluster - In this chapter, we will see how to monitor and debug Kubernetes clusters. You will learn strategies for debugging a misconfigured or misbehaving Kubernetes object. Also, we will see how to monitor clusters and nodes.

Chapter 12: Kubernetes Networking and Services - This chapter will focus on various types of Kubernetes services and will also learn about exposure of Pods inside or outside of the cluster, depending on their declared types. You will also learn Kubernetes networking and network policy.

Chapter 13: Mock Exam 1 - This chapter will cover the first mock exams with solutions. The questions in these exams will give you a real exam-like feel regarding your ability to read and interpret a given question, how to manage time, and validate and debug your own work.

Chapter 14: Mock Exam 2 - This chapter will cover the second mock exam with solutions. The questions in these exams will give you a real exam-like feel regarding your ability to read and interpret a given question, how to manage time, and validate and debug your own work.

Code Bundle and Coloured Images

Please follow the link to download the

Code Bundle and the Coloured Images of the book:

https://rebrand.ly/p199ivx

The code bundle for the book is also hosted on GitHub at In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a

general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

1. Introduction to Kubernetes, Docker, and Minikube

Introduction

Structure

Objectives

Software development, deployment, and evolution

Deploying applications on a physical server

Deploying applications in Virtual Machines

Deploying applications in containers

Evolution of microservices

Introducing Docker

Introducing Kubernetes

Architecture of Kubernetes

kube-apiserver

kube-controller-manager

kube-scheduler

etcd

Cluster nodes

Kubelet

kube-proxy

Container runtime

Benefits of Kubernetes

Installing Docker

Installing minikube

Practice test

Docker

Kubernetes

Conclusion

2. What, Why, and How of CKAD Exam

Introduction

Structure

Objectives

Exam overview

Syllabus

Application deployment

Application design and build

Application environment configuration and security

Application observability and maintenance

Service and networking

Exam environment

Time management

Tips and tricks

Conclusion

Points to remember

3. Exploring Pod, Deployment, ReplicaSet, and Namespace

Introduction

Structure

Objectives

Container and Pod concept

Imperative and declarative approach

Creating, running and scheduling of Pods

Namespace

Deployments

ReplicaSet

Creation of deployments

Conclusion

Exam tips and tricks

Practice test core concepts

Solutions

4. Deployment Strategies and Helm Packages

Introduction

Structure

Objectives

Deployment revisit

Types of deployment

Recreate deployment

Rolling update deployment

Autoscaling deployment

Helm

Introducing Helm

Conclusion

Practice test deployment and Helm

Solutions

5. Pod Design and Concepts

Introduction

Structure

Objectives

Review labels and labels selections

Annotations

Job

Cron job

Conclusion

Practice test

Solution

6. Multi-container Pods and Design Patterns

Introduction

Structure

Objectives

Multi-container Pod

Use cases

Sidecar pattern

Ambassador pattern

Adapter pattern

Init container

Conclusion

Questions

Solution

7. Kubernetes Volumes and Persistence

Introduction

Structure

Objectives

Different types of Kubernetes volume types

Pods and deployments volume configuration

Persistent Volume and PersistentVolumeClaim

Configure Pod with PV and PVC

Storage class

Conclusion

Questions

Solution

8. Configuring Volume, Environment Variables, ConfigMap, and Secrets

Introduction

Structure

Objectives

Kubernetes configuration

Environment variables

Define and consume configuration data

ConfigMaps

Secrets

Mounting volume and environment variables to Pods

Conclusion

Questions

Solutions

9. Service Accounts Resource Quota and Security Contexts

Introduction

Structure

Objectives

Define security context

Resource boundary

Resource quota

Service account

Conclusion

Questions

Solutions

10. Liveness and Readiness

Introduction

Structure

Objectives

Health probing

Readiness probe

Liveness probe

Startup probe

Conclusion

Questions

Solutions

11. Monitoring and Debugging of Kubernetes Cluster

Introduction

Structure

Objectives

Debugging Kubernetes cluster and strategies

Monitor Kubernetes cluster and Pods

Monitor events and logs

Observability and monitoring

Debugging and monitoring tips and tricks

Conclusion

Questions

Solutions

12. Kubernetes Networking and Services

Introduction

Structure

Objectives

Kubernetes Services and types

Creating Services

Deployment and Service

Kubernetes networking and network policy

Troubleshooting Service

Conclusion

Questions

Solutions

13. Mock Exam 1

Introduction

Structure

Objectives

Questions

Conclusion

14. Mock Exam 2

Introduction

Structure

Objectives

Questions

Conclusion

Index

CHAPTER 1

Introduction to Kubernetes, Docker, and Minikube

Introduction

In this foundation chapter, we will learn how software development and deployment have changed over the years and dive deep into understanding the world of Container, Docker, and Kubernetes.

We will also learn about how to set up your minikube. minikube is a tool that allows you to run a single-node Kubernetes cluster locally on your own machine. minikube single node cluster will be used to execute all the practice exercises.

Structure

In this chapter, we will cover the following topics:

Software development, deployment, and evolution

Evolution of microservice

Introducing Docker

Introducing Kubernetes

Architecture of Kubernetes

Benefits of Kubernetes

Installing Docker

Installing minikube

Practice test

Objectives

By the end of this chapter, the reader can understand how software development has changed over the years, the role of containers and virtual machine in the evolution of microservice, Kubernetes architecture. I hope this chapter provides a solid foundation.

Software development, deployment, and evolution

Around 15 years back, if our team wanted to deploy an application or run a service, we had to provision a whole new physical server. Back then, if we were developing an application such as a J2EE application, the team needed to work with the purchasing department to order physical hardware. It would take 4-8 weeks for the hardware to arrive, then the IT team would install OS with the required binaries/software updates and hand over the server to the project team. Only then the project team could deploy the application and initiate user testing before roll-out to production. So, from project inception to actual production, it used to take anywhere between 4-6 months. It was a frustrating, expensive, non-efficient, and time-consuming process. The scaling was costly and difficult, as it entitled buying more hardware.

Cut short to today; we can provision and spin up a new server and deploy applications within a few hours. All the provisioning activities can be conducted using a self-help portal, and there is no dependency on the IT team. Over time, the industry has evolved with technologies such as virtualization and containers. Let us try to understand this evolution from a physical server to virtualization to containers with the help of examples.

Deploying applications on a physical server

Resources are needed for a service or application to run on a physical server. The resources are Central Processing Unit Input/Output and disk These resources are managed by the operating system, as shown in the following figure: there is a hardware layer of the physical server with resources. On the hardware, the operating system is loaded. The application runs on the OS, which helps manage the resource utilization for the application.

In Figure we can see how the resources are managed by the operating system:

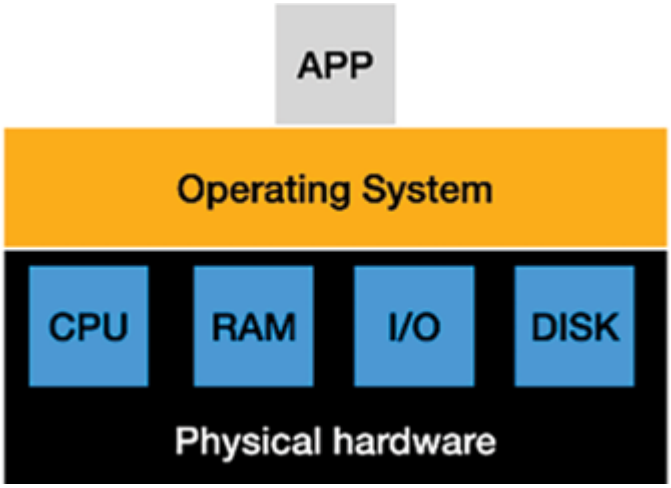


Figure 1.1: Physical

Typically, the physical server architecture is single-tenant; that is, you only have one application running on a physical server. If multiple

applications are running on a physical server, one application may consume most of the compute resources and impact other applications running on the same server. This is a common problem of shared resources without strong boundaries. Hence, a physical server runs only one application in most cases, thus not efficiently utilizing the physical server resources.

Moreover, most of the time, nobody knew the performance requirements of the new application. Procuring a new server would take several months, forcing the IT department to make guesses while choosing the model and size of the server to buy.

Thus, IT department used to buy big fast servers to run the applications, and in most instances, these servers were underutilized. It was a total waste of capital resources and money.

Deploying applications in Virtual Machines

Sometime around the mid-2000s, Virtual Machines came into existence, where each server with VM technology could run multiple applications. The virtual machine was a game changer. With the advent of technologies such as VMware and hypervisor, organizations can utilize physical servers more efficiently. They can now allocate, add, and remove resources must faster. So, there was a drastic improvement in how servers were procured for applications, as new servers were no longer procured for every new application. With VM technologies, multiple applications can reside on a single server application, and the same physical server can be shared across multiple applications.

With a virtual machine, you have the same physical server with all the resources, but instead of the server operating system, there is a hypervisor such as vSphere or Hyper-V loaded to it. The Hypervisor is where you create your virtual machine.

In Figure each VM has its own virtual device, such as a virtual CPU and virtual RAM. On top of these virtual hardware, you load a guest operating system and the applications. Refer to the following figure:

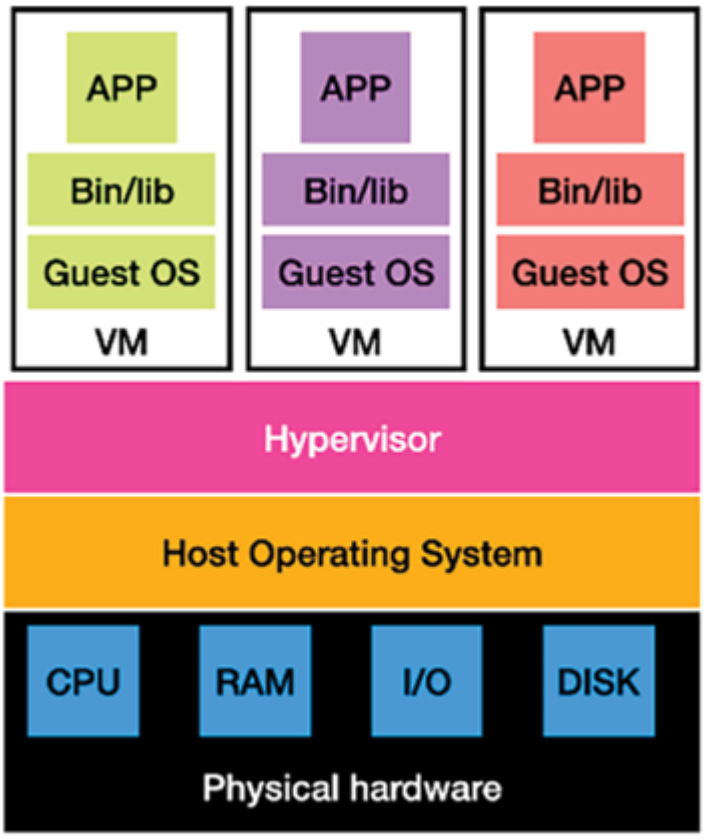


Figure 1.2: Virtual machine

Application services are deployed to each of these virtual machines, so there is a tight boundary and no sharing of the resources. The benefits of virtualization are obvious; instead of having just one application per server, you can now run several guest operating systems and a handful of applications with the same physical hardware.

Deploying applications in containers

So how are containers different from VMs? In short, containers virtualize the operating system instead of the hardware. Containers offer enhanced portability and efficiency by bundling code and dependencies together. Functioning as an abstraction at the application layer, they provide a seamless way to package and run software applications in any environment.

Multiple containers can run on the same machine and share the operating system kernel with other containers. Each of these containers runs as isolated processes. Compared to VMs, containers are only megabytes in size and take just seconds to start, versus gigabytes and minutes for a VM.

In the following figure, containers sit on top of a physical server and its host operating system (either Linux or Windows):

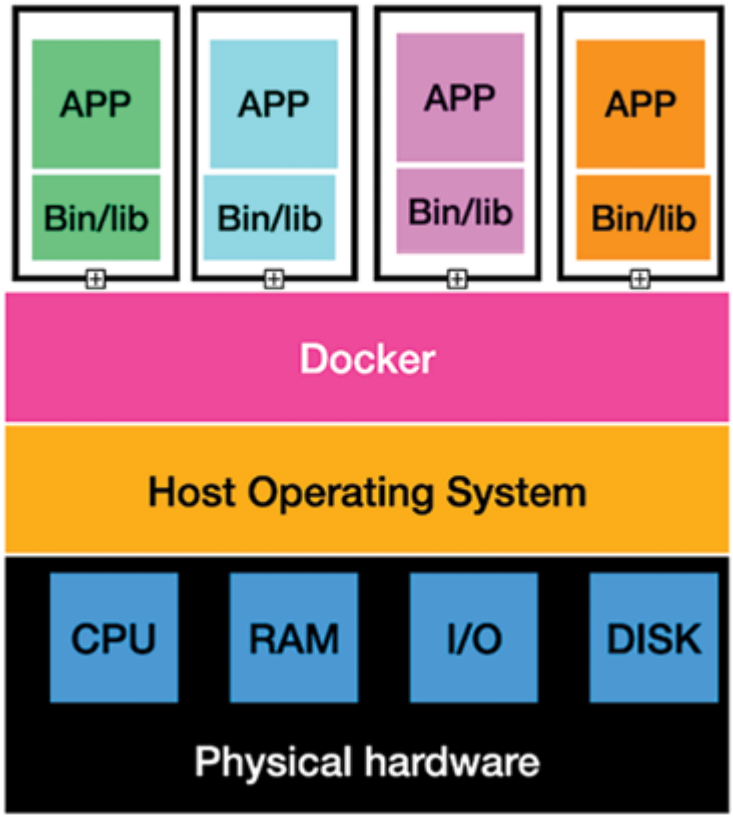


Figure 1.3: Container

Docker is a tool that uses containers to make the creation, deployment, and running of applications a lot easier. It binds the application and its dependencies inside a container.

Evolution of microservices

In the previous sections, we looked at how the software industry has evolved from physical servers to virtual machines to containers. This evolution has also impacted the software architecture, its deployment, and its development.

During the era of physical servers, most of the applications were built as monolithic applications. In monolithic applications, all components must be developed, deployed, and managed as one entity. Any change in one of the application's components required the redeployment of the whole application.

To handle the increase in load on a monolithic system, you could either vertically scale the system by adding more CPUs and memory or other server components. You can also scale horizontally, that is, by setting up additional servers and running multiple copies of the application. Scaling out horizontally has its own challenges, and you will need to re-architect the application if you want to scale out certain components of the application. The components of the applications are tightly coupled to the scale-out. Most of the time, you end up scaling out the whole components of the application, even though the intent was to scale only certain components. So, the solution was to split up this complex monolithic application into independently deployable components called microservices. Each microservice runs independently and communicates with other services through well-defined APIs.

Moreover, since these microservices are standalone processes, they can be independently deployed and scaled. We have seen that the physical server is good for deploying and running monolithic applications, but not the microservice. So, what are the best options to deploy the microservice applications? VMs are a good option for running the application with multiple processes. Each dedicated component can run on its own dedicated VM instance. But as the components become smaller, they pose new challenges to VM. As VMs start to grow, managing them increases the admin overheard.

Since there was a container boom, the adoption of microservices architectures for application development also evolved. Containers allowed applications to be broken down into their smallest parts or services, which served a single purpose. These independent services are not monolithic units and could be developed and deployed independently regardless of each other. For example, let us say you have an online shopping app like Amazon or The application might have a search bar, a shopping cart, a pay button, and so on. Each of those functionalities/services can exist in its own container. Thus, if the search bar or payment button fails due to high load, it does not bring the whole thing down.

Introducing Docker

While container technologies have been around for a long time, they have now become more widely known with the rise of the Docker container platform. The Docker container platform allows you to quickly build, deploy, and test applications. These standardized units that Docker packages the software into are called Containers, which have everything the software needs to run, including system tools, libraries, code, and runtime.

Some concepts of Docker that you should be aware of are as follows:

Container images: You package your application and its environment into a Docker-based container image. This image will inform how a Container is instantiated, determining which software components will run and how. One can also create an image from a container and share these specifications with someone else, so that the app runs the same way on a developer's laptop as it would in production.

Registries: A Docker registry is a repository that stores your Docker images and facilitates helping by easily sharing those images between different people and computers. When you build your image, you can either run it on the computer you have built it on or push (upload) the image to a registry and then pull (download) it on another computer and run it there. Certain registries are public, such as Docker Hub, allowing anyone to pull images from it, while others are private, only accessible to certain people or machines.

Containers: This describes a virtual environment that bundles the application code with all the binaries and libraries required to run an application. Since the Container includes all its dependencies, you do not have to install anything on the host operating system. A running Container is a process running on the Docker host, but it is completely isolated from the host and all other processes running on it. The process is also resource-constrained, meaning it can only access and use the amount of resources (CPU, RAM, and so on) that are allocated to it.

Introducing Kubernetes

As we talked about in the earlier section, in a microservice architecture, there will be n number of services and each of these services can live in their own Container. For example in case of a simple web application, you may have one container for the web frontend, one container for the backend APIs, and another for data storage. If you start seeing some traffic on the web application, you can spin up more instances of your web application components.

Though containers allow you to scale, managing these operations at scale can be complicated. When operating at scale, you must coordinate behavior for the containers and between the Containers. This is when an orchestration engine such as Kubernetes comes into play. Kubernetes is an open-source orchestration system for automating containerized applications' deployment, scaling, and management.

Google was the first company to realize that as the number of deployable application components in your system grows, it becomes harder to manage them all. Over the years, Google developed an internal system called Borg and later a new system called Omega that helped application developers and system administrators manage those thousands of applications and services. In addition to simplifying the development and management, it helped them achieve much higher utilization of their infrastructure, which is important when your organization is as large.

After keeping Borg and Omega secret for a decade, in 2014, Google introduced Kubernetes, an open-source system based on the experience gained through Borg, Omega, and other internal Google systems.

Architecture of Kubernetes

Let us try to understand the overview of Kubernetes and see how it works.

In Kubernetes architecture, there are different nodes, and the most important node in the Kubernetes cluster is what we call the Control/Master

Therefore, a Kubernetes architecture has a Control node and a number of cluster nodes or worker nodes. Master nodes are responsible for maintaining the state of the Kubernetes cluster, whereas worker nodes are responsible for executing your Docker Containers.

Let us get into the meat of the Kubernetes architecture. The Master node is the system that maintains a record of all Kubernetes objects. It continuously manages object states, responding to changes in the cluster. It also works to make the actual state of system objects match the desired state.

The Control/Master plane is made up of below major components:

Kube-apiserver

Kube-controller-manager

Kube-scheduler

etcd

kube-apiserver

Kubernetes' most important component is a REST API called Essentially, this is an API that exposes all Kubernetes features. The kube-apiserver provides APIs, to support lifecycle orchestration (scaling, updates, and so on) for different applications. This API server also acts as the gateway to the cluster, so it must be accessible to clients from outside the cluster.

kube-controller-manager

The controller manager is a daemon that runs the core control loops. It tries to maintain the actual state of the cluster with the one described in the and so there are no differences between the states.

kube-scheduler

As the name suggests, it is a component that schedules. It is responsible for electing a worker node out of those available, to run a newly created Pod. The scheduler is responsible for scheduling containers across the cluster nodes; it considers various constraints, such as resource limitations or guarantees affinity and anti-affinity specifications. It is okay if these terms sound new now. We will look into the details later.

etcd

It is necessary for kube-apiserver to store the state of the cluster somewhere, such as the number of containers created, on which machines, the names of the Pods, which Docker images they use, and more. To achieve that, it uses the etcd database. etcd database is a consistent and highly available key-value store, used as Kubernetes' backing store for all cluster data.

Cluster nodes

Cluster nodes are machines that run containers and are managed by master nodes or control plane of the cluster.

The Cluster nodes are made up of three major components:

Kubelet

kube-proxy

Container runtime

Kubelet

Kubelet is an agent that runs on each node in the cluster, and it ensures that containers are running in a Pod. Kubelet is constantly running HTTP requests against kube-apiserver to retrieve Pods it has to launch. By default, every 20 seconds, the kubelet runs a GET request against the kube-apiserver component, to list the Pods created on etcd that are destined to it. If containers are not created by Kubernetes, they are not managed by the

kube-proxy

An important part of Kubernetes is networking. kube-proxy is a network proxy that runs on each node in your cluster, and the kube-proxy implements part of the Kubernetes Service concept. On each node, kube-proxy maintains network rules and these network rules allow communication to your Pods from network sessions; inside or outside your cluster.

Container runtime

The container runtime component is responsible for running containers. Kubernetes is not limited to Docker; it can manage other container engines, such as rkt container. However, in this book, we will be using Kubernetes with Docker, which is the most common setup.

Benefits of Kubernetes

In modern architecture, most applications are designed as geographically distributed applications, that serve services over the API network. This is different from the traditional architecture, where we just used to deploy an application to a physical server or VM and used to point the DNS to it. Kubernetes is a tool to build and deploy reliable, scalable, distributed applications.

Let us look at some of its benefits:

Speed: Speed in this context is how fast a business can roll out new features to a production application. Back in the day, if you wanted to update the application/software or push out new features, you had to look out for downtime, typically during late nights or weekends when the traffic is low. In today's world, to remain competitive, businesses want to roll out changes to the end user/customer frequently. Hence, there is a need for continuous deployment and pushing new features to production without downtime and waiting for weekends. Some of the core features of Kubernetes such as declarative, self-healing immutability, help it to enable the speed.

Declarative: In Kubernetes, everything is a declarative configuration object that represents desired state of system. In a traditional system, you define a series of instruction whereas in Kubernetes, you just need to define the desired state and Kubernetes will ensure the state is reached.

The declarative configuration lets the user define what state the system should be in and if it is less error prone. This makes rollbacks fairly easy for Kubernetes, which is impossible with imperative traditional configurations.

Self-healing: When Kubernetes receives a declarative configuration, it not only ensures that the desired configuration state is achieved one time, but throughout the life span of the desired configuration of the application. For example, suppose you have set the desired replica of a component or service as three. In that case, Kubernetes not only ensures that there are three replicas but continuously monitors the same to ensure there are always three replicas of the component. Whenever one replica is deleted, Kubernetes will spin up a new replica.

Installing Docker

This section is all about Docker installation. To install Docker, follow the given instructions.

First, go to the following link for your specific operating system: http://docs.docker.com/engine/installation/

Once the installation is complete, you can try the following commands to see if it is working properly:

The command will display the version of Docker installed on your system, as shown in Figure

docker --version:

```
kishore@ss-MacBook-Pro ~ % docker --version
Docker version 20.10.2, build 2291f61
```

Figure version

docker run This command will download a small test image and run it on the container. This is a simple test to verify if the docker is able to pull the image. Refer to Figure

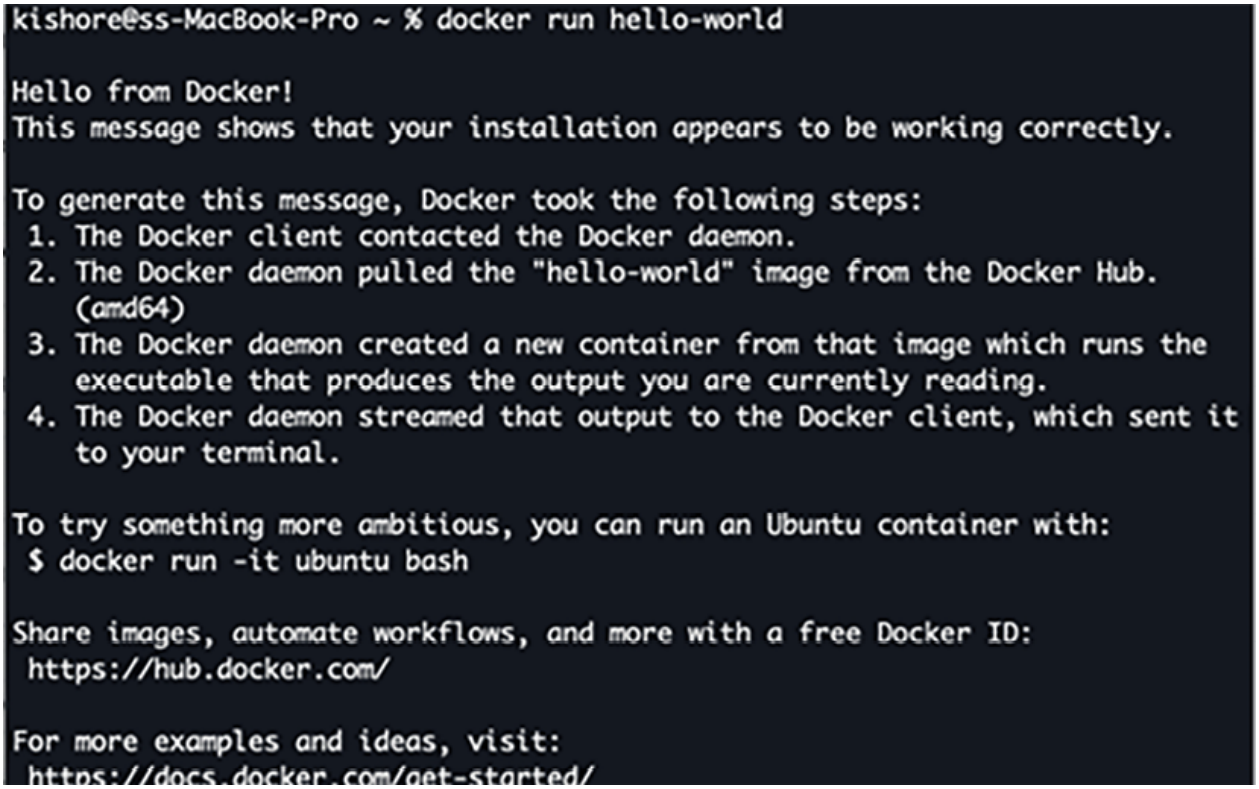

kishore@ss-MacBook-Pro ~ % docker run hello-world
Hello from Docker!
This message shows that your installation appears to be working correctly.
To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
(amd64)
3. The Docker daemon created a new container from that image which runs the
executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
to your terminal.
To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash
Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/
For more examples and ideas, visit:

Figure hello-world

docker This command will display all the images that are currently running on the system. Refer to Figure

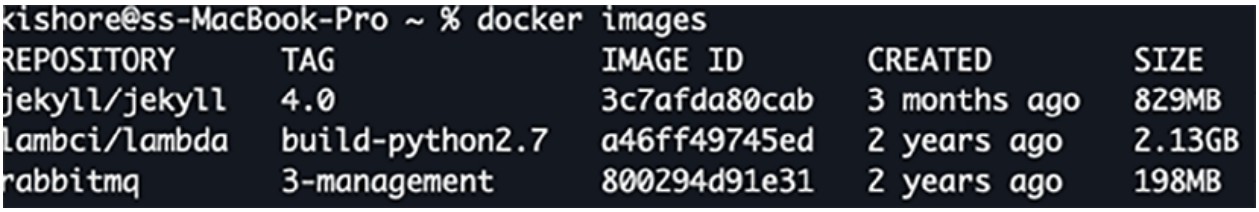

kishore@ss-MacBook-Pro ~ % docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
jekyll/jekyll 4.0 3c7afda80cab 3 months ago 829MB
lambci/lambda build-python2.7 a46ff49745ed 2 years ago 2.13GB
rabbitmq 3-management 800294d91e31 2 years ago 198MB

Figure images

docker ps: This command will list all the running containers on the system.

These are just few of these commands to get started with Docker.

Installing minikube

The next step after the Docker installation is to get Kubernetes up and running. There are two ways to run a Kubernetes cluster, either it can run on your local development machine, on cloud providers providing virtual machines Compute Engine, Amazon EC2, Microsoft and so on), or by using a managed Kubernetes cluster such as Google Kubernetes Engine.

For this book, we recommend you install a single-node Kubernetes cluster. The single-node Kubernetes cluster is called minikube. Single-node minikube cluster should be enough to explore most topics discussed in this book and practice the accompanying exercises.

Moreover, installing minikube is quite easy. It is a single binary that needs to be downloaded and put onto your path. It is available for Mac OS, Linux, and Windows. To install it, the best place to start is to go to the minikube repository on GitHub: From here you will find instructions to install minikube as per your operating system.

After installing minikube, some of the basic commands you can use to get started with in the Kubernetes cluster, are as follows:

minikube start: This command will start a new Kubernetes cluster. It will download and start a virtual machine with Kubernetes installed.

minikube status: The command will display the status of the minikube cluster.

minikube stop: The command will stop the minikube cluster.

minikube delete: This command will delete the minikube cluster.

These are just a few commands of the many minikube commands available to manage the Kubernetes minikube cluster.

Practice test

This foundation chapter’s objective is to get the docker and minikube installed. Ensure you are able to execute the following commands, once installed successfully.

Docker

docker –version

docker run hello-world

Kubernetes

minikube start

minikube status

minikube stop

minikube delete

Conclusion

In this foundation chapter, we learnt how software development and deployment have changed over the years and tried to understand the world of Container, Docker, and Kubernetes.

In the next chapter, we will do a deep dive into the CKAD exam.

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 2

What, Why, and How of CKAD Exam

Introduction

This chapter is about the CKAD exam, syllabus, exam environment, and tips and tricks related to the exam. Kubernetes and container technology with new software architecture patterns like microservices have become extremely popular in the tech industry in the last few years. For container orchestration solutions, Kubernetes has become the first choice. As Kubernetes has become popular and standard for container orchestration, there is a growing demand for developers who can design, build and manage applications using Kubernetes. Developers can confirm to potential employers that they have the skills and knowledge required to work with Kubernetes by obtaining the CKAD certification, which they can use to show their expertise in the Kubernetes domain.

Structure

This chapter will cover the following topics:

Exam overview

Syllabus

Exam environment

Time management

Tips and tricks

Objectives

This chapter aims to give an overview of the CKAD exam. The exam syllabus, how the exam is conducted, and also tips and tricks that will help you during the preparation for the actual exam are also discussed.

Exam overview

The Certified Kubernetes Application Developer exam is a technical certification offered by Cloud Native Computing Foundation They want to emphasize the developers who wish to demonstrate their skills in building, deploying, and managing cloud-native applications on Kubernetes. Before we get into the details of each domain, we want to emphasize that the CKAD exam is all about speed. The exam lasts 2 hours, and you need to solve 19 questions within the time frame. The Kubernetes version operating in the CKAD exam environment when this chapter was written is version 1.27. The percentile for passing this exam is 66%. The exam differs from other cloud exams, mostly theory-based multiple-choice exam formats. In contrast, the CKAD exam is an online, hands-on, performance-based exam that requires you to perform tasks in a command-line environment.

If you pass the CKAD exam, you will receive a digital certificate and a badge from the CNCF demonstrating your Kubernetes application development proficiency. The certification is valid for two years. Overall, the CKAD exam is challenging and provides a rewarding certification that validates your skills in building and managing cloud-native applications on Kubernetes.

Syllabus

The curriculum of the CKAD exam covers the following topics, and each topic of the exam carries a different weight:

Application deployment: 20%

Application design and build: 20%

Application observability and maintenance: 15%

Application environment, configuration, and security: 25%

Services and networking: 20%

You can find more about these in detail through the following link

https://training.linuxfoundation.org/certification/certified-kubernetes-application-developer-ckad/

Now let us look into the details of each domain and identify what they consist of. If you are new to Kubernetes, you will hear lots of new terms, so do not worry at this stage and stay calm. We will dive deep into details in the coming chapters.

Application deployment

This section is all about managing Pods and deployments. This section of the exam covers the core general structure of Kubernetes like deployment, Pods, and its representation in Yet Another Mark-Up Language You should be aware of and will be tested on creating, deleting, and modifying the Kubernetes object like Pod via YAML or To master this section, you must know how Kubernetes objects are created, deleted, and modified. You will also learn how to manage these via the command line. Also, you will be tested on the Helm package management.

Application design and build

In this exam section, you will learn some advanced concepts and configuration options of Kubernetes objects. You need to understand multi-container Pods concepts and their various patterns like the sidecar, adapter, and ambassador patterns. You also need to know reading and writing data via volume, and you will be tested on how to create and configure them.

Application environment configuration and security

This section will test you on authentication, authorization, and admission control. You will also be tested on how to define resource requirements, limits, and quotas. How to configure ConfigMaps and secrets. It would be best to define a resource limit (maximum and minimum) for container and service accounts.

Application observability and maintenance

In this section, you will be tested on how to implement probes and health checks, and how to monitor and debug Kubernetes applications. You will be tested on the liveness and readiness of probe topics. You should also know the difference between them, which probes to use under which scenario, and how to debug various probes. You should know the command to check the logs of a Pods and save them in an external file and how to monitor application usage.

Service and networking

In this section, you will be tested on the basic understanding of network policies, how applications can be accessed via services, and how to expose applications using ingress rules. You need to understand how to create different types of services and their port-mapping mechanism and you should be familiar with configuring network policies and access rules for incoming and outgoing traffic.

Exam environment

The CKAD exam is a performance-based exam conducted on a Linux-based command-line environment with access to Kubernetes clusters. The exam environment includes a web-based terminal and a Kubernetes cluster with one or more nodes. You will be given tasks to complete during the exam in a command-line environment- The tasks will require creating, modifying, and managing Kubernetes resources such as Pods, deployments, services, and You will be given a list of resources you can refer to during the exam, such as the Kubernetes documentation, kubectl cheat sheet, and Kubernetes API reference. It is important to note that the CKAD exam is timed, and you will have two hours to complete 19 tasks. Therefore, time management is crucial during the exam, and you should practice completing tasks quickly and efficiently.

The CKAD exam environment is designed to test your hands-on skills in Kubernetes application development. You should be familiar with using kubectl and working in a command-line environment.

The proctor would inform you of the time left, typically at 1 hour 30 mins and 10 minutes before your time would be up. He will ask you to click the end exam button when the test is over, and then you will be presented with a survey form so that you can give your opinion (optional). You need to wait for 36 hours before the results are made available.

If you pass, you get a congratulatory note and a link to download your CKAD certificate. A happy moment indeed.

Time management

The hands-on exam has no multiple-choice questions, so time management is key. Use the two hours wisely and tackle the questions with high weightage points first. On average candidate will get 6.3 minutes per question. Once you have completed the high points questions and crossed the 66% pass mark, attempt the rest. Throughout the chapter, there will be highlighted strategies that will help you complete your tasks, for example, using the kubectl imperative approach rather than the declarative one.

Tips and tricks

We know that speed is an important factor in cracking this exam. Below are some of the tips and tricks that you use can use during the exam:

During the exam, you can use the Kubectl command to explain things, as shown in the following code and Figure

kubectl explain pod.spec

```
kishore@ss-MacBook-Pro ~ % kubectl explain pod.spec
KIND:     Pod
VERSION:  v1

RESOURCE: spec <Object>

DESCRIPTION:
     Specification of the desired behavior of the pod. More info:
     https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-an
d-status

     PodSpec is a description of a pod.

FIELDS:
   activeDeadlineSeconds        <integer>
     Optional duration in seconds the pod may be active on the node relative to
     StartTime before the system will actively try to mark it failed and kill
     associated containers. Value must be a positive integer.

   affinity     <Object>
     If specified, the pod's scheduling constraints

   automountServiceAccountToken <boolean>
     AutomountServiceAccountToken indicates whether a service account token
```

Figure 2.1: Explain Pod command

You can use the Kubectl option for help as shown in the following code and Figure

kubectl run po --help | grep -i requests

```
Create and run a particular image in a pod.

Examples:
  # Start a nginx pod
  kubectl run nginx --image=nginx

  # Start a hazelcast pod and let the container expose port 5701
  kubectl run hazelcast --image=hazelcast/hazelcast --port=5701

  # Start a hazelcast pod and set environment variables "DNS_DOMAIN=cluster" and
"POD_NAMESPACE=default" in the container
  kubectl run hazelcast --image=hazelcast/hazelcast --env="DNS_DOMAIN=cluster"
--env="POD_NAMESPACE=default"

  # Start a hazelcast pod and set labels "app=hazelcast" and "env=prod" in the
container
  kubectl run hazelcast --image=hazelcast/hazelcast
--labels="app=hazelcast,env=prod"

  # Dry run; print the corresponding API objects without creating them
  kubectl run nginx --image=nginx --dry-run=client

  # Start a nginx pod, but overload the spec with a partial set of values parsed
from JSON
  kubectl run nginx --image=nginx --overrides='{ "apiVersion": "v1", "spec": {
```

Figure 2.2: kubectl help option

Be aware of basic Vim commands, which will help you save time. Jumping to a specific line, copying a certain number of lines from the cursor, going to the end of the line, and deleting a character are a few command examples.

The exam has multiple clusters and namespaces provisioned. So, before the start of the question, please ensure you are in the correct cluster and namespaces. You will be provided a command to switch between clusters in the exam.

Conclusion

This chapter is all about the CKAD exam. Hope this chapter has helped you understand the CKAD exam better.

In the next chapter, we will dive deeply into each exam topic.

Points to remember

Once you have a good grasp of topics covered in the exam, ensure you practice.

Practice and effective time management is the key to cracking the CKAD exam.

At the end of each chapter in this book, there will be practice questions; ensure you practice those thoroughly.

There are also trips and tricks at the end of each chapter. There is no hands-on exercise for this chapter.

CHAPTER 3

Exploring Pod, Deployment, ReplicaSet, and Namespace

Introduction

In this chapter, we will start learning Kubernetes resource objects, Pods, deployments, and namespace. We will also discuss the basic structure of Kubernetes primitives and how to interact with Kubernetes objects via the command line–based client, By the end of this chapter, you will understand how to create Kubernetes objects imperatively and declaratively with basic configuration. We will end the chapters with some exercises.

Structure

In this chapter, we will cover the following topics:

Container and Pod concept

Imperative and declarative approach

Creating, running and scheduling of Pods

Namespace

Deployments

Creation of deployments

ReplicaSet

Objectives

By the end of this chapter, you will understand and know how to manage Kubernetes core objects of Pods, deployment , namespace and replicaset

Container and Pod concept

Container is a self-contained ready-to-run application. In Chapter Introduction to Kubernetes, Docker and minikube we introduced container and how it differs from virtual machine. We also looked at docker, which is a very important player in the field of containers. Now let me introduce you to Pod.

What is Pod? At a high level, it is nothing but an abstraction of the server. In traditional IT, the server provides everything for the application to run. In Kubernetes container, Pod provides everything to run the application. In Kubernetes, Pod is the minimum entity Kubernetes talks to. Note that Kubernetes does not manage the container directly; it manages Pod, which holds one or many containers. Another point to remember is that Pod has its IP address in a Kubernetes cluster.

We have already discussed the minikube setup. Refer to Chapter Introduction to Kubernetes, Docker and minikube. Get the setup done. Now if you have yet to configure any Pod in the minikube cluster and try to query the cluster with the number of Pods, you will see the message No resource found in default Namespace is the area or space allocated for Kubernetes objects, acting like a folder. We will be looking at namespace in more detail further in the chapter.

Run the command using the kubectl command as shown below:

$kubectl get pods

The above command lists all Pods in the cluster.

Refer to the following Figure

Figure 3.1: Command to get pods in a cluster

Imperative and declarative approach

We can create Pod using the kubectl run command. For example, if you want to create a nginx Pod. The command to create nginx Pod is shown below:

$kubectl run nginx --image=nginx --restart=Never --port=8080

The above command is imperative to creating and starting a nginx Pod.

Refer to the following Figure

Figure 3.2: Create and run Pod command

And now, if you run the get pods command, you will see nginx Pod is running in cluster:

$kubectl get pods

The above command lists all Pods in the cluster, showing the newly created nginx Pod.

Refer to the following Figure

Figure 3.3: Command to get Pods in a cluster

Other than Pod name, you also see the READY header. The READY is indicating 1/1. It means that a Pod is expecting one container to be available, and it does have one container. The STATUS of the Pod is shown running, which means Pod is in a running state. The header AGE indicates that Pod has not been restarted and has been running for

Let us dissect the run command:

$kubectl run nginx --image=nginx --restart=Never --port=8080

Refer to the following Table

Table 3.1: Run command details

The other option to create the pod is using a manifest file; the following manifest file is saved as

apiVersion: v1

kind: Pod

metadata:

```
  name: ymlnginx

spec:

  containers:

  - image: nginx

    name: nginx

    ports:

    - containerPort: 8081

  restartPolicy: Never
```

Using create or apply command can create the Pod from the manifest file as shown below:

$kubectl create -f

The above command creates a Pod using the data in

Refer to the following Figure

```
kishore@ss-MacBook-Pro ~ % kubectl create -f pod.yaml
pod/ymlnginx created
```

Figure 3.4: Command to create Pod in cluster using data in YAML

Now, if we do kubectl get pods we see both the pods. One created imperatively using kubectl command line and another created using YAML:

$kubectl get pods

The above command lists all Pods in the cluster, it will show both the Pods created via imperative and declarative method.

Refer to the following Figure

```
kishore@ss-MacBook-Pro ~ % kubectl get pods
NAME       READY   STATUS    RESTARTS   AGE
nginx      1/1     Running   0          24m
ymlnginx   1/1     Running   0          4m59s
```

Figure 3.5: Command to get Pod in the cluster

The question is when to use a manifest file vs kubectl to create Kubernetes object like Pod. Using kubectl to create a Kubernetes object is much faster than creating a manifest file and then creating the object. Since the CKAD exam is about speed, the kubectl imperative approach is a better option.

Sometimes, it is better to opt for a hybrid technique where you create an object using the imperative command and then create a manifest file. The dry-run command option, as shown below, will execute the run command without creating the Kubernetes objects, and the YAML definition will be saved in podymlfile.yml -o yaml provides insight to all pod parameters:

$ kubectl run backend --image=nginx --restart=Never --port=80 \

-o yaml --dry-run=client > podymlfile.yaml

The above command is an example of a hybrid technique. The Pod definition will be saved in podymlfile.yaml

backslash(\) is the shell escape character, which is used to continue a command on the next line. When your command is continuning to the next line please use backslash.

Refer to the following Figure

```
kishore@ss-MacBook-Pro ~ % kubectl run backend --image=nginx --restart=Never --port=80 \
  -o yaml --dry-run=client > podymlfile.yaml
```

Figure 3.6: Command to save the Pod definition in YAML without creating it

With kubectl run command, you can only create a pod object. Kubectl create command is used to create another Kubernetes object imperatively.

Creating, running and scheduling of Pods

To become more efficient with the kubectl command, you can use the kubectl resources alias rather than the whole name, for example, for Pods, you can use

$ kubectl get po

The above command uses the short name po instead of pod.

Refer to the following Figure

Figure 3.7: Command to get Pod from cluster using short name

To get the list of short names of Kubernetes objects from you can run the following command:

$ kubectl api-resources

The above command prints supported API resources.

Refer to the following Figure

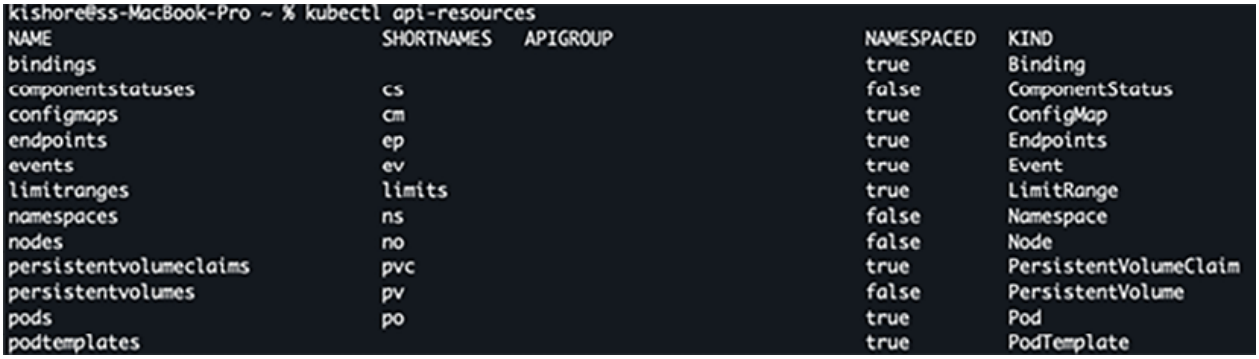

kishore@ss-MacBook-Pro ~ % kubectl api-resources
NAME SHORTNAMES APIGROUP NAMESPACED KIND
bindings true Binding
componentstatuses cs false ComponentStatus
configmaps cm true ConfigMap
endpoints ep true Endpoints
events ev true Event
limitranges limits true LimitRange
namespaces ns false Namespace
nodes no false Node
persistentvolumeclaims pvc true PersistentVolumeClaim
persistentvolumes pv false PersistentVolume
pods po true Pod
podtemplates true PodTemplate

Figure 3.8: Command to print API resources

If you want to delete a Pod, use delete command:

$ kubectl delete po ymlnginx

Above command is to delete Pod from cluster, in the above case Pod with name ymlnginx will be deleted.

Refer to the following Figure

Figure 3.9: Command to delete Pod from the cluster

kubectl describes Pods and shows the details about Pod, including information about the container running within it:

$kubectl describe POD nginx

The description will print debug information of given resources like in the above command example, it will print information of Pod nginx.

Refer to the following Figure

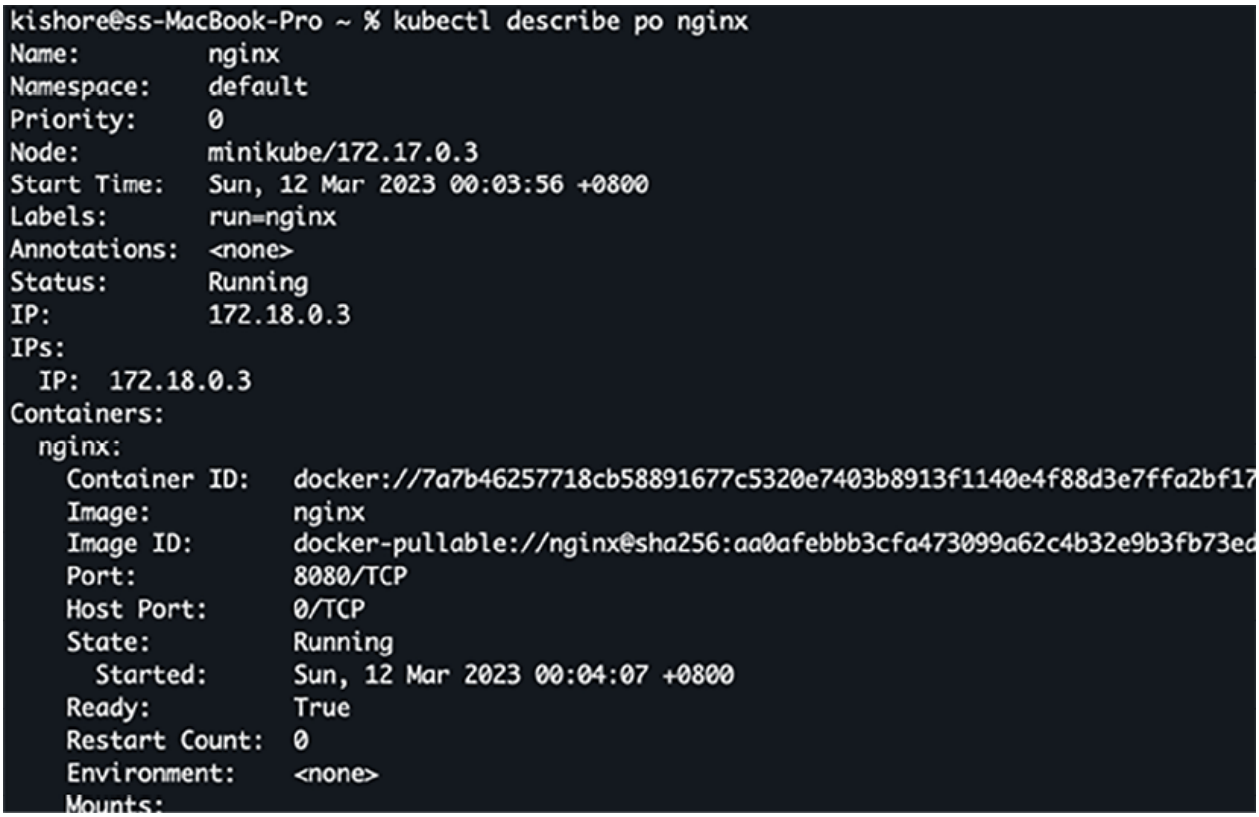


Figure 3.10: Print debug information of the Pod

The describe command provides a human-readable description. The description provides various information related to Pods, like which namespace and Node the Pod runs. The IP address, image details, restart counts, and other details of Pod. These details are helpful in getting the current state of the Kubernetes object. So, describe command is quite a useful command to query the state of Kubernetes object.

Kubectl has a help command that you can use to learn about various offerings with examples:

$kubectl run -h

The above command is an example of how you use the help command to get more details of the run command.

Refer to the following Figure

```
kishore@ss-MacBook-Pro ~ % kubectl run -h
Create and run a particular image in a pod.

Examples:
  # Start a nginx pod
  kubectl run nginx --image=nginx

  # Start a hazelcast pod and let the container expose port 5701
  kubectl run hazelcast --image=hazelcast/hazelcast --port=5701

  # Start a hazelcast pod and set environment variables "DNS_DOMAIN=cluster" and
"POD_NAMESPACE=default" in the container
  kubectl run hazelcast --image=hazelcast/hazelcast --env="DNS_DOMAIN=cluster"
--env="POD_NAMESPACE=default"

  # Start a hazelcast pod and set labels "app=hazelcast" and "env=prod" in the container
  kubectl run hazelcast --image=hazelcast/hazelcast --labels="app=hazelcast,env=prod"

  # Dry run; print the corresponding API objects without creating them
  kubectl run nginx --image=nginx --dry-run=client

  # Start a nginx pod, but overload the spec with a partial set of values parsed from JSON
  kubectl run nginx --image=nginx --overrides='{ "apiVersion": "v1", "spec": { ... } }'
```

Figure 3.11: Print help information

To access the logs of Pod, execute the following command:

$kubectl logs nginx

The above command prints logs from Pod

Refer to the following Figure

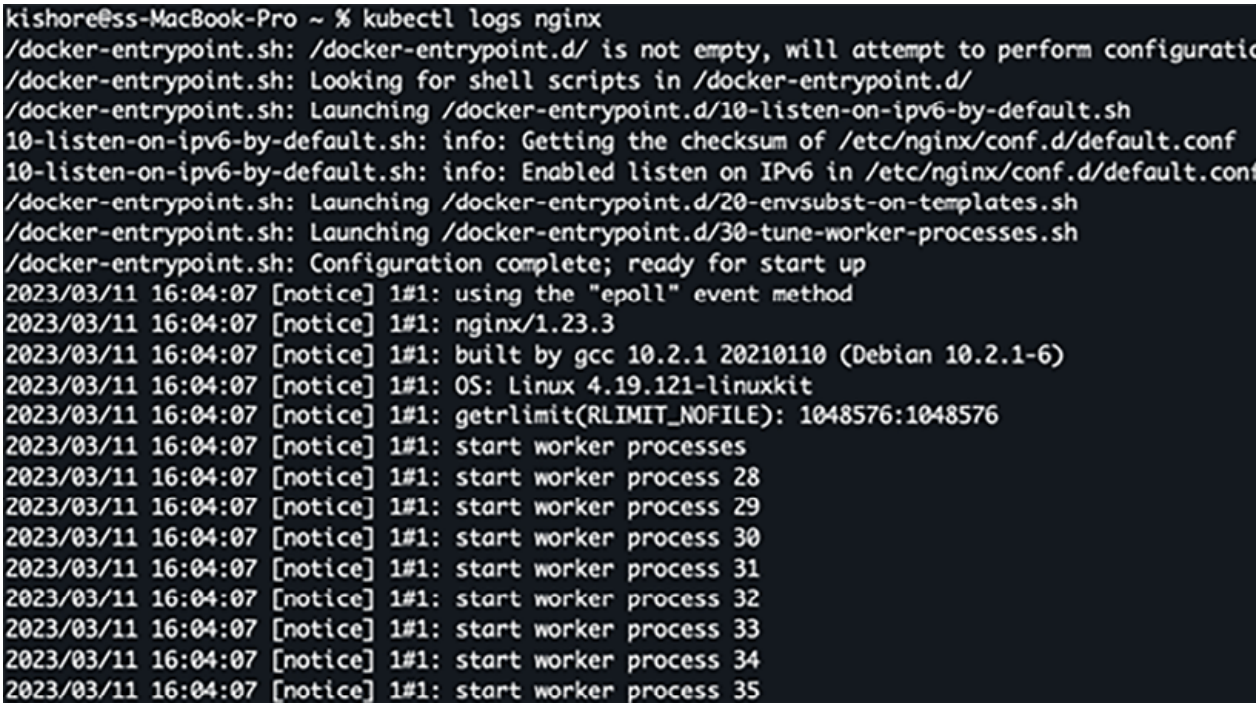

kishore@ss-MacBook-Pro ~ % kubectl logs nginx
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuratio
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2023/03/11 16:04:07 [notice] 1#1: using the "epoll" event method
2023/03/11 16:04:07 [notice] 1#1: nginx/1.23.3
2023/03/11 16:04:07 [notice] 1#1: built by gcc 10.2.1 20210110 (Debian 10.2.1-6)
2023/03/11 16:04:07 [notice] 1#1: OS: Linux 4.19.121-linuxkit
2023/03/11 16:04:07 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2023/03/11 16:04:07 [notice] 1#1: start worker processes
2023/03/11 16:04:07 [notice] 1#1: start worker process 28
2023/03/11 16:04:07 [notice] 1#1: start worker process 29
2023/03/11 16:04:07 [notice] 1#1: start worker process 30
2023/03/11 16:04:07 [notice] 1#1: start worker process 31
2023/03/11 16:04:07 [notice] 1#1: start worker process 32
2023/03/11 16:04:07 [notice] 1#1: start worker process 33
2023/03/11 16:04:07 [notice] 1#1: start worker process 34
2023/03/11 16:04:07 [notice] 1#1: start worker process 35

Figure 3.12: Print logs from Pod

These are the log of the container, and it provides information as to what is going on inside a container and is useful for debugging. Note that the logs container only renders out for current container. If a container is restarted for some reason, and you want to see the logs from the previous container use -p command line option.

Sometimes there is a need to log into the container and execute the command, you can use exec to open a shell in the container and explore it interactively.

Execute the command in a container:

$kubectl exec -it nginx -- /bin/sh

The above command is used to get an interactive shell on Pod.

Refer to the following Figure

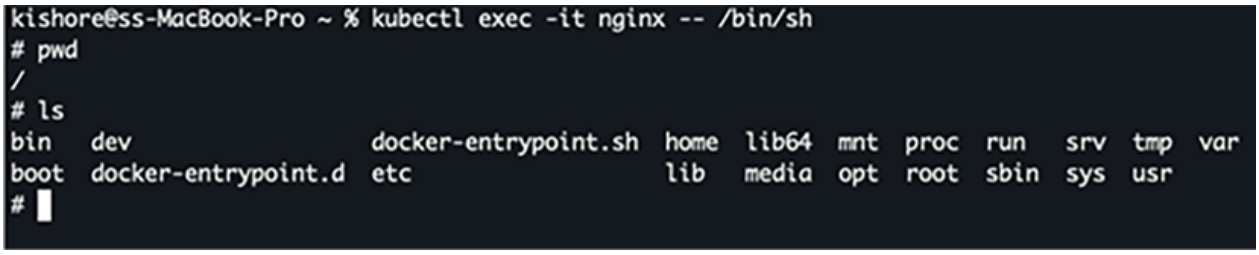

kishore@ss-MacBook-Pro ~ % kubectl exec -it nginx -- /bin/sh
pwd
/
ls
bin dev docker-entrypoint.sh home lib64 mnt proc run srv tmp var
boot docker-entrypoint.d etc lib media opt root sbin sys usr
#

Figure 3.13: Command to get an interactive shell on Pod

Using the preceding command, you log into the nginx container and try out various commands like

The two dashes - - separates the exec command with the command you want to run inside the container:

$kubectl exec nginx -- ls

Refer to the following Figure

```
kishore@ss-MacBook-Pro ~ % kubectl exec nginx -- ls
bin
boot
dev
docker-entrypoint.d
docker-entrypoint.sh
etc
home
lib
lib64
media
```

Figure 3.14: Command to run inside the container

If you do not want to execute a single command in the Pod and check its output, you can remove it (interactive) and add the command after two dashes -

Environment variable: You can set the environment variable for the container that runs the Pod. Setting up environment variables is a way to make Pod configurable at runtime.

apiVersion: v1

kind: Pod

metadata:

  name: FEAPP

spec:

```
containers:

- image: nginx

  name:  nginxfe

  env:

  - name: morning_greetings

    value: Goodmorning
```

Here the env name morning_greetings environment variable is defined with values

Namespace

Kubernetes namespace is a way for different teams to share a cluster. It provides isolation and helps to divide the cluster into logic units. The resources within each of the namespaces can communicate if needed.

If you type get namespaces, you will see set of namespaces that Kubernetes cluster by default started.

$kubectl get namespace

The above command gets all namespace in the cluster.

Refer to the following Figure

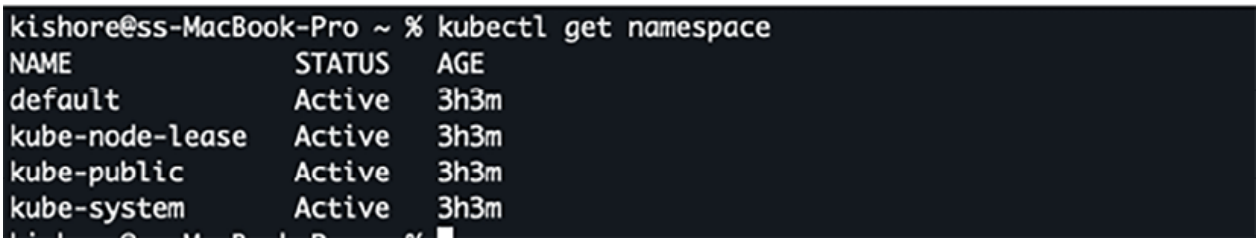


Figure 3.15: Command to get all namespace in the cluster

Those namespaces with prefix kube- are created by Kubernetes system. The command to create a new namespace, say test is:

$kubectl create namespace test

The above command is used to create a namespace test in the cluster.

Refer to the following Figure

```
kishore@ss-MacBook-Pro ~ % kubectl create namespace test
namespace/test created
```

Figure 3.16: Command to create namespace in the cluster

So, now you will see the test as one of the namespaces:

$kubectl get namespace

The above command will show the newly created namespace test.

Refer to the following Figure

```
kishore@ss-MacBook-Pro ~ % kubectl get namespace
NAME              STATUS   AGE
default           Active   3h6m
kube-node-lease   Active   3h6m
kube-public       Active   3h6m
kube-system       Active   3h6m
test              Active   48s
```

Figure 3.17: Command to get namespace in the cluster

Create a new Pod under namespace see how we have used short name n for namespace

$kubectl run hazelcast --image=hazelcast/hazelcast -n test

The above command uses -n shortform for

Refer to the following Figure

```
kishore@ss-MacBook-Pro ~ % kubectl run hazelcast --image=hazelcast/hazelcast -n test
pod/hazelcast created
```

Figure 3.18: Use of shortname for namespace

If you execute get pod command in the default namespace, you will not see the newly created Pod in other namespace, say for example, test.

Refer to the following Figure

```
kishore@ss-MacBook-Pro ~ % kubectl get po
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          176m
```

Figure 3.19: Command to get Pods in the cluster in default namespace

If you query for the Pods under the test name space, you will see the newly created Pod.

Refer to the following Figure

Figure 3.20: Command to get Pods in the cluster under namespace test

Deleting a namespace: $kubectl delete test, deletes the namespace

Deleting a namespace deletes all the objects inside, so be careful when you delete it, as it will have a cascading effect.

Deployments

In a production environment, you rarely run Pod as a standalone; you will always run it as deployments. Deployments offer you reliability and scalability to the applications running on Pod. Scalability means how the applications running on Pods can handle an increase or decrease in changing demand of the application usage. Also, deployment offers reliability, that is, it ensures the Pod is always running, serving the requests, and will restart the Pod if needed and ensure status is maintained if anything goes wrong.

ReplicaSet

Deployment is using Kubernetes primitive ReplicaSet to maintain the number of Pods. The ReplicaSet ensures replication of a number of Pods with the same configuration.

Refer to the following Figure

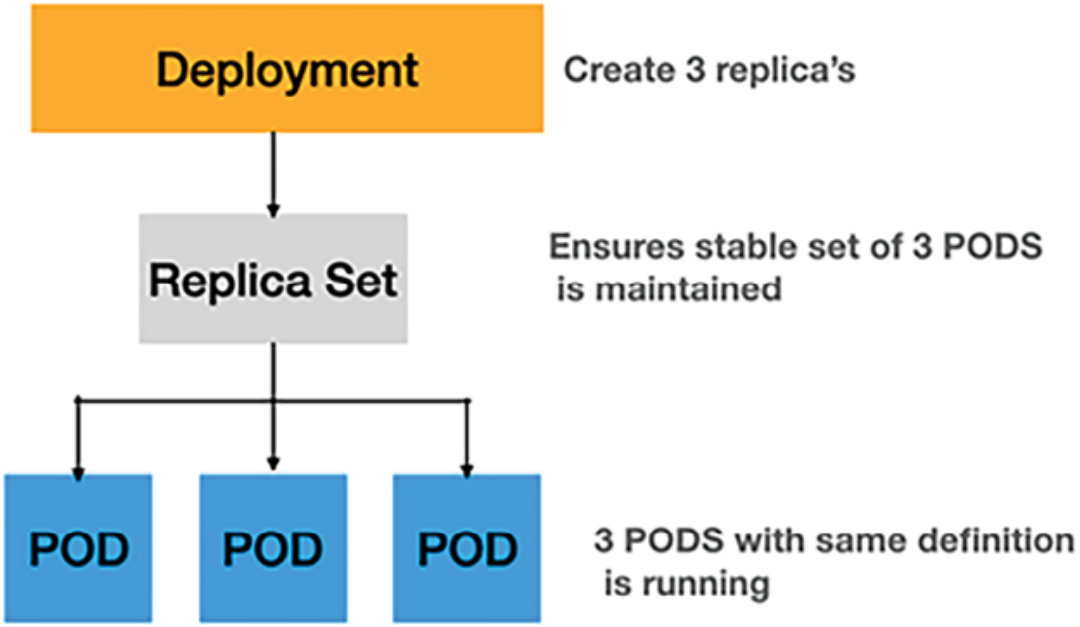


Figure 3.21: Diagram showing relationship of Deployment ReplicaSet and Pods

Creation of deployments

So, to create the deployment, use kubectl create command and mention the replicas of Pod.

$kubectl create deployment nginx-deploy -–image=nginx –-replicas=3

The above command creates deployment with the name with 3 nginx Pods.

Refer to the following Figure

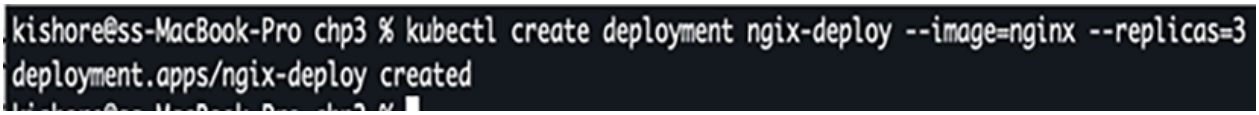

kishore@ss-MacBook-Pro chp3 % kubectl create deployment ngix-deploy --image=nginx --replicas=3
deployment.apps/ngix-deploy created

Figure 3.22: Command create deployment of three ngix Pods

Execute kubectl command to get the Pod and deployments.

$kubectl get deployment

The preceding commands get the deployments in the cluster.

Refer to the following Figure

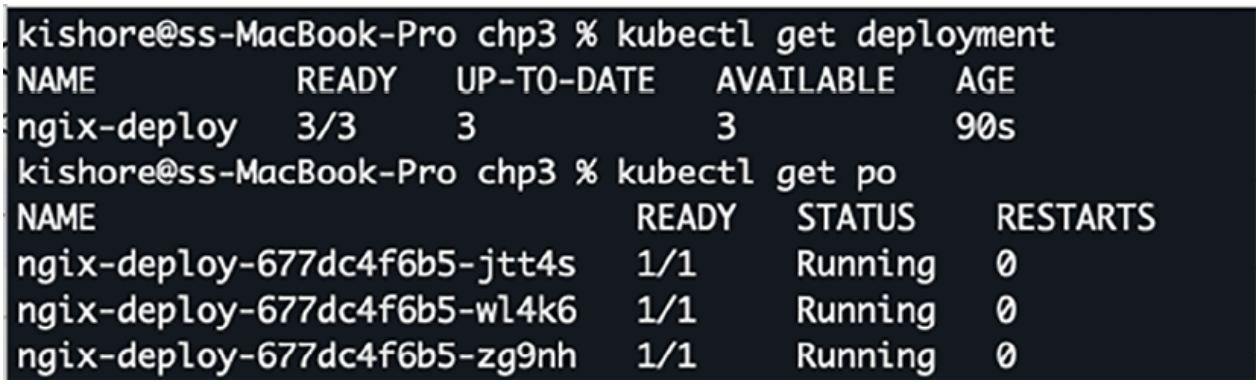

kishore@ss-MacBook-Pro chp3 % kubectl get deployment
NAME READY UP-TO-DATE AVAILABLE AGE
ngix-deploy 3/3 3 3 90s
kishore@ss-MacBook-Pro chp3 % kubectl get po
NAME READY STATUS RESTARTS
ngix-deploy-677dc4f6b5-jtt4s 1/1 Running 0
ngix-deploy-677dc4f6b5-wl4k6 1/1 Running 0
ngix-deploy-677dc4f6b5-zg9nh 1/1 Running 0

Figure 3.23: Command showing the deployments and Pods of a given cluster

If you delete a Pod using the kubectl delete pod command, the replica set will spin up a new Pod and ensure the 3 count is maintained.

The below command is used to delete the deployment:

$kubectl delete deployment nginx-deploy

Conclusion

In this chapter, you learned how to manage the Pods namespace and were introduced to deployment. You also learned imperative and declarative ways to create Kubernetes objects.

Exam tips and tricks

In Kubernetes, objects can be created using a declarative way or an imperative way. In the CKAD exam, time is essence, so try to use the imperative way of using kubectl to create objects

Any time you are stuck in the exam, you can use the kubectl help (command to get more info on the command.

Refer to the following Figure

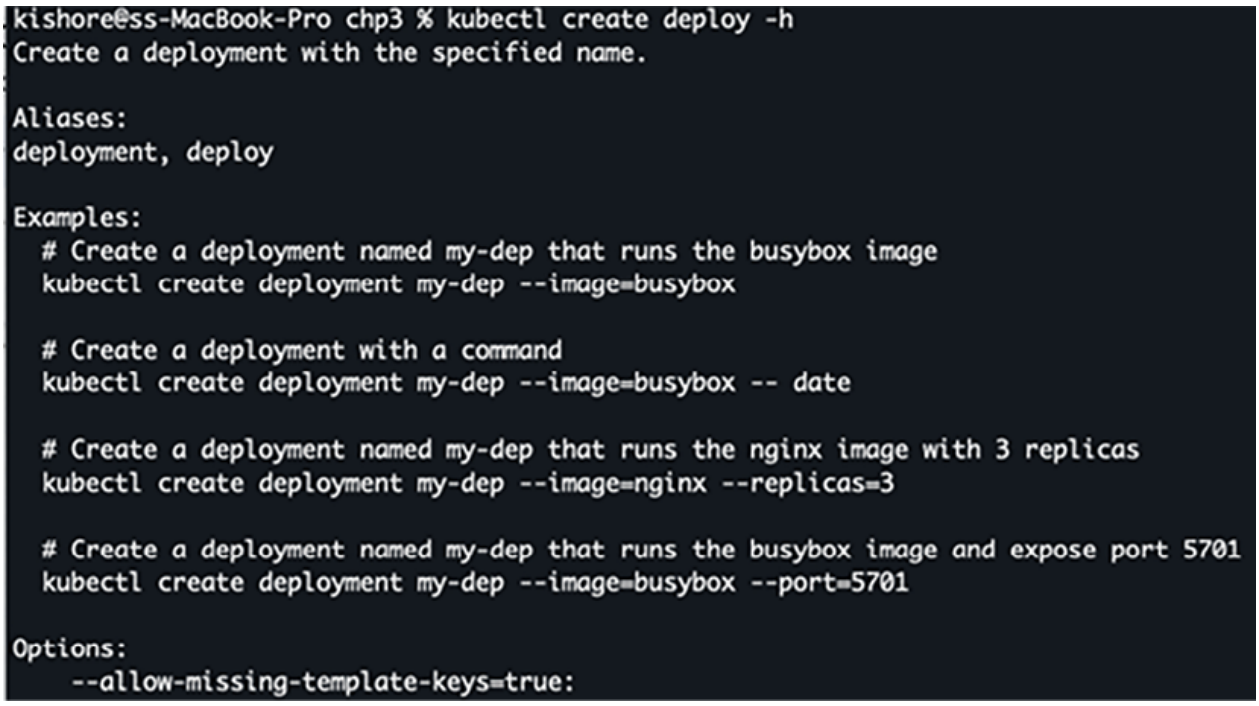

kishore@ss-MacBook-Pro chp3 % kubectl create deploy -h
Create a deployment with the specified name.
Aliases:
deployment, deploy
Examples:
Create a deployment named my-dep that runs the busybox image
kubectl create deployment my-dep --image=busybox
Create a deployment with a command
kubectl create deployment my-dep --image=busybox -- date
Create a deployment named my-dep that runs the nginx image with 3 replicas
kubectl create deployment my-dep --image=nginx --replicas=3
Create a deployment named my-dep that runs the busybox image and expose port 5701
kubectl create deployment my-dep --image=busybox --port=5701
Options:
--allow-missing-template-keys=true:

Figure 3.24: Command showing how to use help

Inspect the object with the kubectl get or kubectl describe commands to check the Pod's status and configuration.

Practice test core concepts

Creating Pod: Create a Pod named test1, which should be launched from nginx version 1.17.10

Creating Pod in a namespace: Create a pod named test2, which should be launched with image hazelcast. The Pod should live in namespace name2. Expose the container port 5701

For test2 Pod created, access the log. Get the Pod details including its IP's

For the test2 pod, open a shell for the hazelcast container and inspect the contents of the current directory ls

Create a pod with the name of test3. The Pod should be launched from an image of nginx. The name of the container should be chp3cont. The container should execute the print timestamp and sleep every 10 seconds

Create a deployment named deploy1. The deployment should be launched from the nginx image and replicas of 3 Pods. Delete the namespace name2 and delete the Pods

Solutions

Solution to question 1:

You can either use a declarative approach using YAML or an imperative approach using

Using kubectl (imperative approach)

```
$ kubectl run test1 --image=nginx:1.17.10
```

Using YAML, (declarative approach)

create a file with the test1.yaml

```
apiVersion: v1

kind: Pod

metadata:

  name: test1

spec:
```

```
  containers:

  - name: nginx

    image: nginx:1.17.10
```

Create pod from yaml

```
$ kubectl create -f test1.yaml
```

Solution to question 2:

You can either use a declarative approach using YAML or an imperative approach using

Using kubectl (imperative approach)

Create namespace

```
$ kubectl create namespace name2
```

Create POD in namespace name2

```
$ kubectl run test2  --image=hazelcast/hazelcast --port=5701 --namespace=name2
```

Using YAML (declarative approach)

create a file with the test1.yaml

```
apiVersion: v1

kind: Pod

metadata:

  name: test2

spec:

  containers:

  - name: hazel

    image: hazelcast/hazelcast

  Ports:


  - containerPort: 5701
```

Create pod from yaml

```
$ kubectl create -f test1.yaml -n name2
```

Solution to question 3:

Accessing logs of test2 pod under namespace name2

$ kubectl logs test2 -n name2

$ kubectl get pod test2 --namespace=name2 -o wide

Solution to question 4:

Accessing the content under the container using command

$ kubectl exec -it test2 --namespace=name2 -- ls

Solution to question 5:

Create YAML sleeppod.yml

apiVersion: v1

kind: Pod

metadata:

  name: test3

spec:

```
  containers:


  - args:
    - /bin/sh
    - -c
    - while true; do date; sleep 10; done
    image: nginx
    name: chp3cont
```

Create pod from yaml

```
$ kubectl create -f sleeppod.yml
```

Solution to question 6:

Create deployments

```
$kubectl create deploy deploy1 --image=nginx --replicas=3
```

Delete namespace name2

$kubectl delete namespace name2

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 4

Deployment Strategies and Helm Packages

Introduction

In this chapter, we will learn about deployment strategies, types of deployment, and Helm packages. Helm packages were recently introduced to the CKAD exam, and we will end the chapter with some exercises. So, let us get started.

Structure

In this chapter, we will cover the following topics:

Deployment revisit

Types of deployment

Rolling deployment

Recreate deployment

Helm

Objectives

By the end of this chapter, you will understand the types of deployment strategies and how Helm is used in the Kubernetes world.

Deployment revisit

In Chapter Core Concepts - Pods, Deployments, ReplicaSet, and Namespace, we got introduced to deployment and saw the relationship between deployment and ReplicaSet. We also saw how you could create a deployment in a cluster.

Like Pod, we also saw that you could query the deployment details using describe commands.

$kubectl describe deploy ngix-deploy

The above command describes deployment in a cluster.

Refer to the following Figure

```
kishore@ss-MacBook-Pro ~ % kubectl describe deploy ngix-deploy
Name:                   ngix-deploy
Namespace:              default
CreationTimestamp:      Sat, 18 Mar 2023 23:37:22 +0800
Labels:                 app=ngix-deploy
Annotations:            deployment.kubernetes.io/revision: 1
Selector:               app=ngix-deploy
Replicas:               3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=ngix-deploy
  Containers:
   nginx:
    Image:        nginx
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Progressing    True    NewReplicaSetAvailable
```

Figure 4.1: Describe deployment in a cluster

Out of various details that describe command prints, it refers to three replicas in the above case. You also see new terms like strategy type, a rolling update, and rolling update strategy details like max unavailable and max surge. Let us look at these terms in more detail.

Types of deployment

There are two types of deployment, recreate and rolling update, which you should be familiar with for the exam. Let us try to understand this with an example, say you have a blog-site application running in Pod and deployed in Kubernetes architecture.

Refer to the following Figure

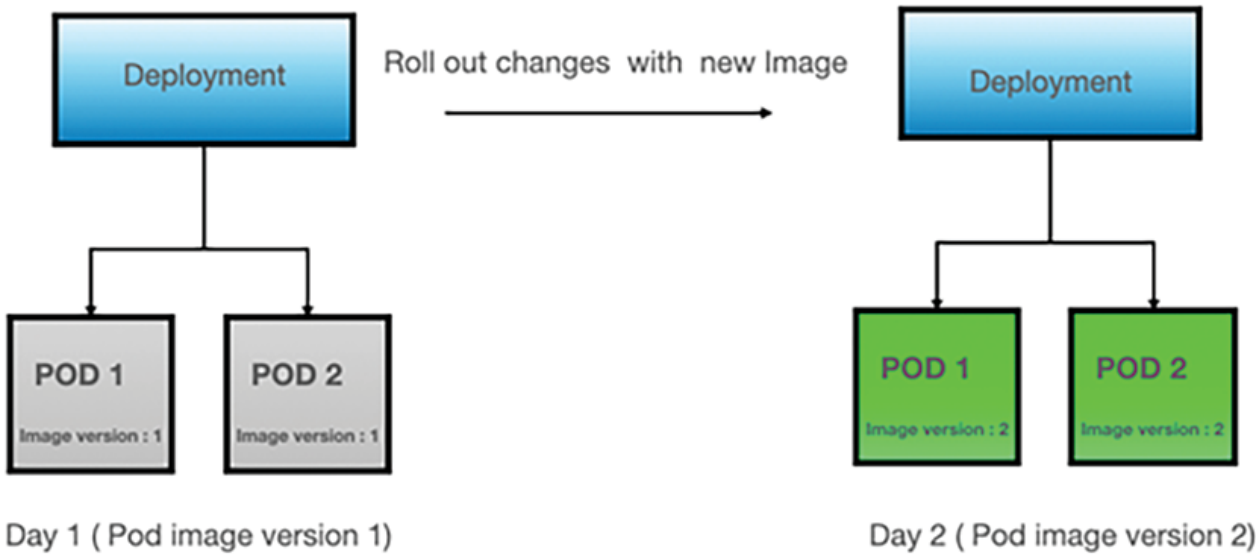


Figure 4.2: Rolling out changes

From the preceding diagram, Pod1 and Pod2 are currently deployed with version 1.0 images of the blog application. Your blog site has become popular, and you would like to enable additional features, say a weekly newsletter, that the paid subscriber can only access. Pod1, and Pod2 must run the version 2.0 image with new features for these changes to be effective. So,

how to roll out these changes and update the Pod to the new version 2.0 image? There are two ways you can push version 2.0 changes:

Delete all the existing Pods with version 1.0 images and then bring up Pods with version 2.0 images.

Sequentially add new Pods with version 2.0 images and, at the same time, gradually delete the old Pods with version 1.0 images.

With option 1, when you delete all the existing Pods with old images and then bring up Pods with new images, there will be a period of application downtime between the switch of images where users cannot access your application.

In the real world, end application users do not access Pods directly, but the user requests the application, which is routed via Kubernetes Service. Kubernetes Service acts as an endpoint and forwards user requests to a set of Pods. We will look at the Kubernetes service object in detail in Chapter Services and

As shown in the figure below, blog user access services which call Pods. Pods are rolled out with new images, which have the changes:

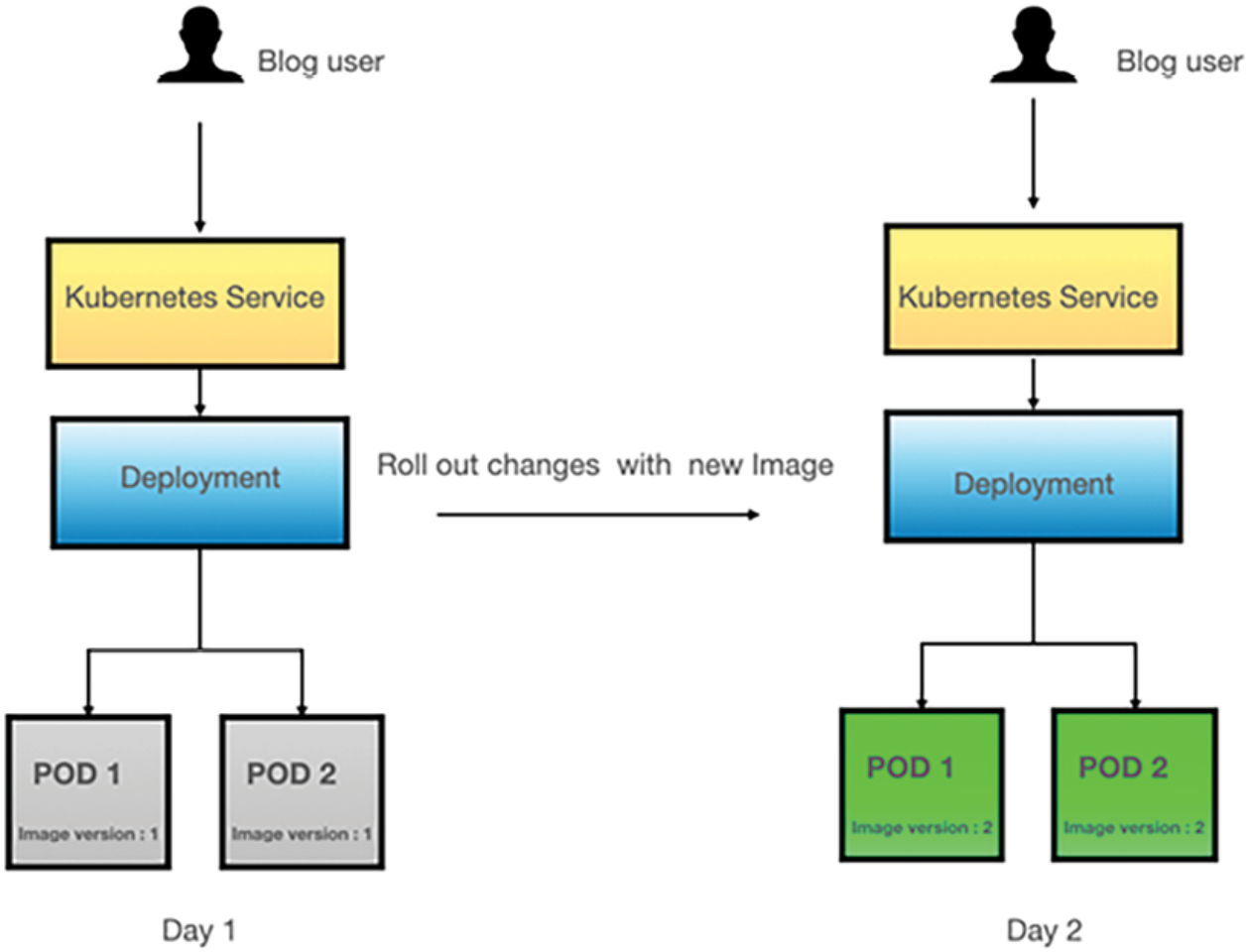


Figure 4.3: End-user access Blog application Day1 and Day 2

As discussed earlier, two ways to push version 2 changes are controlled by strategy type in the deployment definition. These are the Rolling deployment strategy and Recreate deployment strategy.

Recreate deployment

In this deployment strategy, all Pods will be killed. There will be downtime where the application cannot be accessed, and a new set of Pods will be recreated. Refer to the following figure:

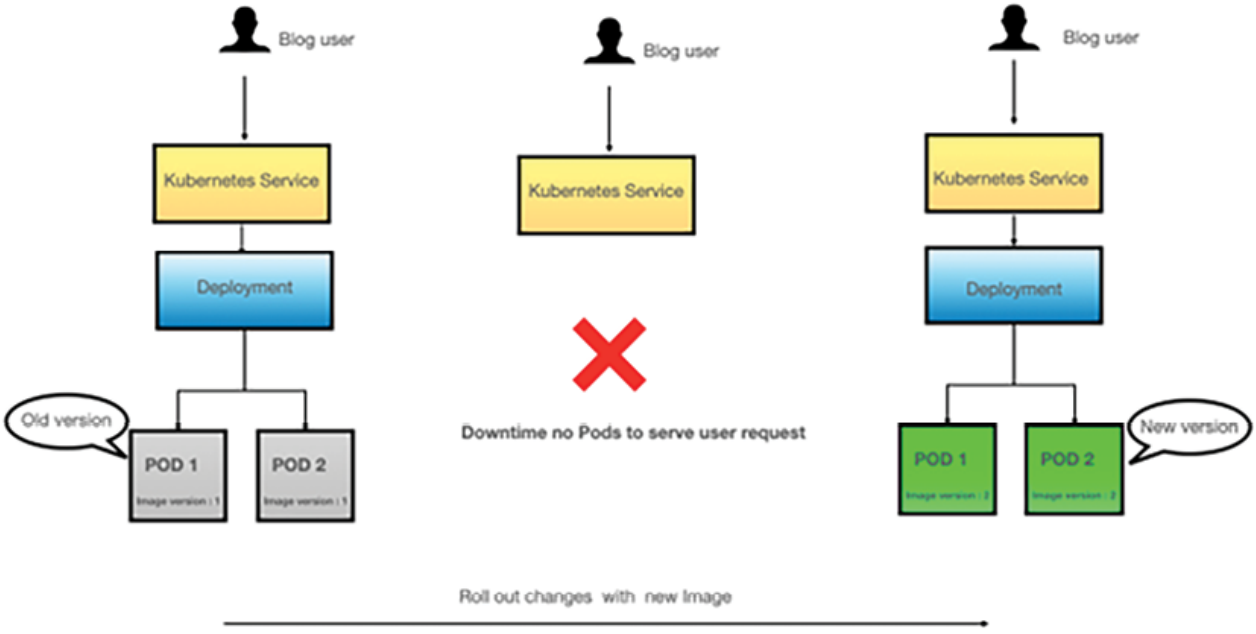


Figure 4.4: Recreate deployment Strategy(with application downtime)

In the deployment definition YAML, you can define the strategy type to recreate (as highlighted below):

apiVersion: apps/v1

kind: Deployment

```
metadata:

  name: nginx-deploy

labels:

    app: nginx-deploy

spec:


  replicas: 3

  selector:

    matchLabels:

      app: nginx-deploy

  strategy:

      type: Recreate

  template:

    metadata:

      labels:
```

```
      app: nginx-deploy
  spec:
    containers:
    - name: nginx
      image: nginx:1.14.2
```

Save the above file as Create the deployment using the YAML file.

kubectl create -f nginx-deploy.yaml

The above command is a declarative approach to create a deployment with strategy type Recreate.

Refer to the following Figure

kishore@ss-MacBook-Pro deployment % kubectl create -f nginx-deploy.yaml
deployment.apps/nginx-deploy created
kishore@ss-MacBook-Pro deployment %

Figure 4.5: Recreate deployment strategy (with application downtime)

kubectl get deploy -o wide

The above command provides wider details to the deployment.

Refer to the following Figure

kishore@ss-MacBook-Pro deployment % kubectl get deploy -o wide
NAME READY UP-TO-DATE AVAILABLE AGE CONTAINERS IMAGES SELECTOR
nginx-deploy 3/3 3 3 89s nginx nginx:1.14.2 app=nginx-deploy
kishore@ss-MacBook-Pro deployment %

Figure 4.6: Provide additional details of the deployment

Update the deployment image using the set command; in this case, the nginx is set to image 1.9.5.

kubectl set image deploy/nginx-deploy nginx=nginx:1.9.5

The set command is used to overwrite or set the given cluster. In the above case, it is used to update the image.

Refer to the following Figure

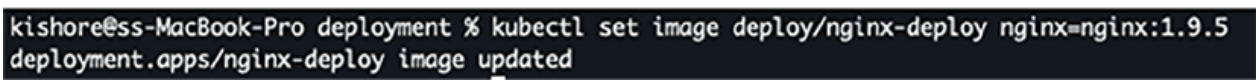

kishore@ss-MacBook-Pro deployment % kubectl set image deploy/nginx-deploy nginx=nginx:1.9.5
deployment.apps/nginx-deploy image updated

Figure 4.7: Set command to update the image

As the deployment strategy type is set to Recreate, the existing Pods will be first killed. Hence if you query for the deployment details, you see the

available Pods as 0. This is the stage of the application downtime that was referred to before.

Refer to the following Figure

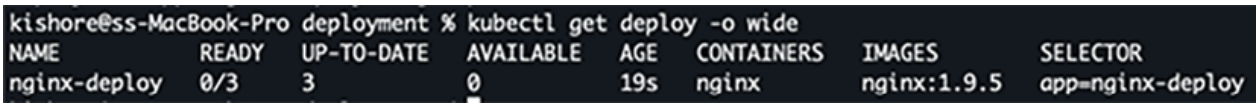

kishore@ss-MacBook-Pro deployment % kubectl get deploy -o wide
NAME READY UP-TO-DATE AVAILABLE AGE CONTAINERS IMAGES SELECTOR
nginx-deploy 0/3 3 0 19s nginx nginx:1.9.5 app=nginx-deploy

Figure 4.8: No Pods to serve request as all the existing Pods are killed before new Pods are rolled out

And once all the Pods are killed, a new set of Pods is recreated with a new nginx image nginx 1.9.5

Refer to the following Figure

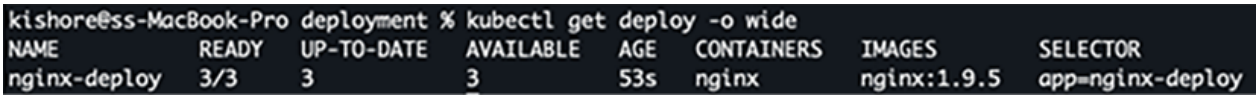

kishore@ss-MacBook-Pro deployment % kubectl get deploy -o wide
NAME READY UP-TO-DATE AVAILABLE AGE CONTAINERS IMAGES SELECTOR
nginx-deploy 3/3 3 3 53s nginx nginx:1.9.5 app=nginx-deploy

Figure 4.9: New set of Pods recreated using updated image

Kubernetes will create versioning whenever Pods is created or updated using deployment.

kubectl rollout history deployment nginx-deploy

The above commands list out previous versions of revisions and configurations.

Refer to the following Figure

```
kishore@ss-MacBook-Pro deployment % kubectl rollout history deployment nginx-deploy
deployment.apps/nginx-deploy
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
```

Figure 4.10: Lists the previous revisions and configurations

In the above deployment, there were two versions; hence there are two revisions in the output.

To update the change-cause column, use --record=true while executing the command.

kubectl set image deploy/nginx-deploy nginx=nginx:latest --record=true

The above commands capture the change cause.

Refer to the following Figure

```
kishore@ss-MacBook-Pro deployment % kubectl set image deploy/nginx-deploy nginx=nginx:latest --record=true
Flag --record has been deprecated, --record will be removed in the future
deployment.apps/nginx-deploy image updated
kishore@ss-MacBook-Pro deployment %
```

Figure 4.11: Command which records the change cause

Now if you again look for revisions. You will see the captured change cause.

Refer to the following Figure

```
kishore@ss-MacBook-Pro deployment % kubectl rollout history deployment nginx-deploy
deployment.apps/nginx-deploy
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
3         kubectl set image deploy/nginx-deploy nginx=nginx:latest --record=true
```

Figure 4.12: Lists the revisions

You can also check the status of the deployment by executing the rollout command as shown below:

kubectl rollout status deployment nginx-deploy

The above command lists out the status of the deployment.

Refer to the following Figure

```
kishore@ss-MacBook-Pro deployment % kubectl rollout status deployment nginx-deploy
deployment "nginx-deploy" successfully rolled out
```

Figure 4.13: Command to list status of the deployment

You can also retrieve information about a revision with the rollout history command, and you must provide the revision number using the

kubectl rollout history deployments nginx-deploy --revision=2

The above command retrieves the information of a particular revision.

Refer to the following Figure

```
kishore@ss-MacBook-Pro deployment % kubectl rollout history deployments nginx-deploy --revision=2
deployment.apps/nginx-deploy with revision #2
Pod Template:
  Labels:       app=nginx-deploy
        pod-template-hash=6b576446b8
  Containers:
   nginx:
    Image:      nginx:1.9.5
    Port:       <none>
    Host Port:  <none>
    Environment:        <none>
    Mounts:     <none>
  Volumes:      <none>
```

Figure 4.14: Command to retrieve information of version

You can also roll back to the previous version using undo command.

So, let us check the current deployment details:

kubectl get deploy -o wide

It is set to the latest image of nginx.

Refer to the following Figure

```
kishore@ss-MacBook-Pro deployment % kubectl get deploy -o wide
NAME           READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS   IMAGES         SELECTOR
nginx-deploy   3/3     3            3           21m   nginx        nginx:latest   app=nginx-deploy
```

Figure 4.15: Command to display current deployment

Let us check the rollout history, as you see, there are three deployment revisions.

Refer to the following Figure

```
kishore@ss-MacBook-Pro deployment % kubectl rollout history deployment nginx-deploy
deployment.apps/nginx-deploy
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
3         kubectl set image deploy/nginx-deploy nginx=nginx:latest --record=true
```

Figure 4.16: Lists the revisions

Let us rollback to the previous version of revision 2:

kubectl rollout undo deployment nginx-deploy --to-revision=2

The above command is the rollback to the previous version.

Refer to the following Figure

```
kishore@ss-MacBook-Pro deployment % kubectl rollout undo deployment nginx-deploy --to-revision=2
deployment.apps/nginx-deploy rolled back
```

Figure 4.17: Rolls back to previous revision

Now if you check the latest version of the deployment, it is rolled back to the previous nginx version.

Refer to the following Figure

```
kishore@ss-MacBook-Pro deployment % kubectl get deploy -o wide
NAME           READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS   IMAGES         SELECTOR
nginx-deploy   3/3     3            3           24m   nginx        nginx:1.9.5    app=nginx-deploy
```

Figure 4.18: Command to display current deployment

If you look at the rollout history, you will find revisions 1,3,4. Kubernetes recognizes that revisions 2 and 4 are the same.

Refer to the following Figure

```
kishore@ss-MacBook-Pro deployment % kubectl rollout history deployment nginx-deploy
deployment.apps/nginx-deploy
REVISION  CHANGE-CAUSE
1         <none>
3         kubectl set image deploy/nginx-deploy nginx=nginx:latest --record=true
4         <none>
```

Figure 4.19: Lists the revisions

Rolling update deployment

This strategy recreates Pods using a rolling update or incremental fashion. In this strategy, Pods are killed and created sequentially, and this strategy will not cause any application downtime. Please refer to the following figure:

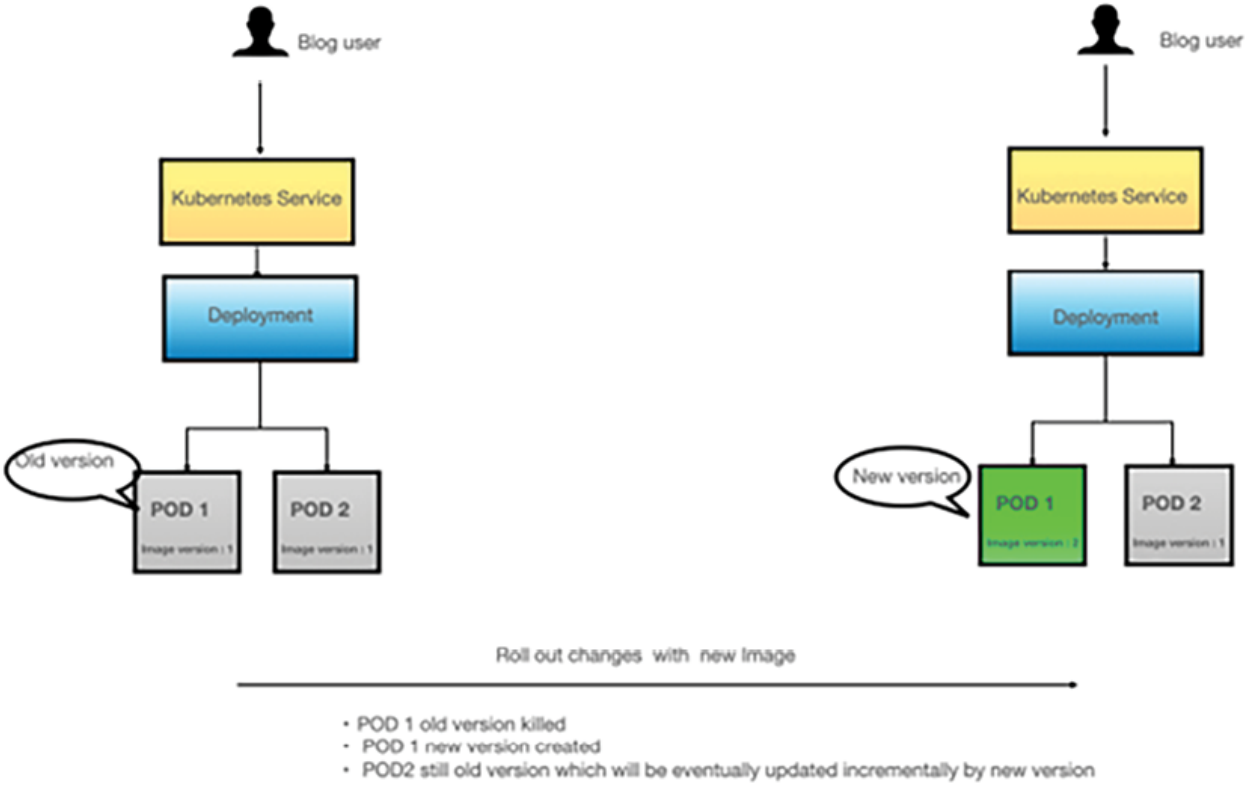


Figure 4.20: Rolling update strategy

In the deployment definition, you can define the strategy type to Rolling update (as highlighted below):

apiVersion: apps/v1

```
kind: Deployment

metadata:

  name: nginx-deploy

  labels:



    app: nginx-deploy

spec:

  replicas: 3

  selector:

    matchLabels:

      app: nginx-deploy

  strategy:

      type: Rollingupdate

      Rollingupdate:

    Maxsurge: 25%
```

```
    Maxunavailabe: 25%

template:

metadata:

      labels:

        app: nginx-deploy

    spec:

      containers:

      - name: nginx



        image: nginx:1.14.2
```

In the Rolling update strategy type, there are two new parameters Maxsurge and

This parameter limits the maximum number of Pods created above the desired number. Assume there are 4 Pods and say the max surge is 25%, then 25% of 4 Pods equals one Pod. This ensures a maximum of 5 Pods is up during the Rolling update strategy update process.

The parameters define the number of Pods that can be unavailable during the Rolling update strategy process. This ensures that a minimum number of Pods are always running.

This parameter ensures ease of deployment and stability during the Rolling update strategy process. The two-parameter default value is 25%.

All the commands we have seen in the Recreate strategy section, like updating the deployment image, checking the deployment version using the rollout command, and undoing deployment to the previous version, apply to the Rolling update strategy also.

Autoscaling deployment

Irrespective of the type of deployment strategy, you can always scale Pods using the scale.

Kubectl scale deployment nginx-deploy --replicas=5

The above command scales the deployment to 5.

Refer to the following Figure

kishore@ss-MacBook-Pro deployment % kubectl get deploy
NAME READY UP-TO-DATE AVAILABLE AGE
nginx-deploy 3/3 3 3 5h19m
kishore@ss-MacBook-Pro deployment % kubectl scale deployment nginx-deploy --replicas=5
deployment.apps/nginx-deploy scaled
kishore@ss-MacBook-Pro deployment % kubectl get deploy
NAME READY UP-TO-DATE AVAILABLE AGE
nginx-deploy 5/5 5 5 5h20m

Figure 4.21: Command to scale live Kubernetes object

As shown above, with scale command, you can change the count of replica attribute of live Kubernetes cluster, as shown in the preceding example deployment nginx-deploy, which had 3 Pods and can be scaled to 5.

In real production Kubernetes world, you seldom use manual methods to scale Pods. In production, Kubernetes cluster auto-scaling is configured, where the Pod is scaled based on memory and CPU utilization metrics.

kubectl autoscale deploy nginx-deploy --cpu-percent=70 --min=2 --max=6

In the above example, we specify a minimum of 2 replicas at any given time, a maximum number of 6 replicas, and the CPU utilization threshold of 70%. Initially, there will be 2 Pods and as the CPU reaches 70% of utilization, the new Pods to the maximum of 6 Pods will be spun up.

Helm

We have seen that deploying and managing applications in Kubernetes is complex, comprising various Kubernetes objects like Pod, deployments, and services to make it work. One needs to manage various YAML files for different Kubernetes objects. So, instead of using various YAML files, a package manager would be good for managing various Kubernetes objects.

Introducing Helm

Helm is a package manager in Kubernetes. In the Kubernetes world, if you want to install any packages, the easiest way is to use Helm. The Helm package allows to install Helm Charts from various Helm Registries.

Helm consists of three things:

Helm tool, which needs to be installed.

Chart, which is the package that you want to install.

Kubernetes cluster, which in our case is minikube.

The first step is to install Helm.

So, we need to fetch the latest version of the Helm binary from the below link:

https://github.com/helm/helm/releases/tag/v3.11.2

Download the latest current Helm release for your operating system, as of writing this book, the latest version is 3.11.2. Once installed, you can check the version of installed Helm using the command:

```
$helm version
```

Charts are the packaging format for Helm. Helm manages Kubernetes resource packages through charts. Helm chart consists of a couple of components, like a description of the package and one or more templates containing Kubernetes manifests files and the values.

Charts can be stored locally, or you can access them from remote Helm repositories.

To create a new chart with a given name, use the following command:

$helm create hello-world

This creates a directory where charts are created. If you check the content of the directory inside the template directory, a few templates for common Kubernetes resources have already been created:

This is the main file containing our chart’s description.

this is the file containing our chart’s default values.

This is the directory where Kubernetes resources are defined as templates.

This is an optional directory that may contain sub-charts

Refer to the following figure for chart structure:

```
kishore@ss-MacBook-Pro hello-world % ls -ltr
total 16
-rw-r--r--   1 kishore  staff  1147 Apr  2 21:04 Chart.yaml
-rw-r--r--   1 kishore  staff  1878 Apr  2 21:04 values.yaml
drwxr-xr-x  10 kishore  staff   320 Apr  2 21:04 templates
drwxr-xr-x   2 kishore  staff    64 Apr  2 21:04 charts
```

Figure 4.22: Chart structure in local machine

Refer to the following figure for files under the template folder:

```
kishore@ss-MacBook-Pro templates % ls -ltr
total 56
-rw-r--r--  1 kishore  staff  2087 Apr  2 21:04 ingress.yaml
-rw-r--r--  1 kishore  staff  1880 Apr  2 21:04 deployment.yaml
-rw-r--r--  1 kishore  staff   373 Apr  2 21:04 service.yaml
-rw-r--r--  1 kishore  staff   328 Apr  2 21:04 serviceaccount.yaml
-rw-r--r--  1 kishore  staff   928 Apr  2 21:04 hpa.yaml
-rw-r--r--  1 kishore  staff  1763 Apr  2 21:04 NOTES.txt
-rw-r--r--  1 kishore  staff  1822 Apr  2 21:04 _helpers.tpl
drwxr-xr-x  3 kishore  staff    96 Apr  2 21:04 tests
```

Figure 4.23: Structure inside the template directory

You can update the deployment.yaml and install the chart into the Kubernetes cluster.

Now you can create your chart as mentioned in the previous section or explore the existing chart in the chart repository like the Bitnami Helm repository. To search chart in this repository, you need to add the Helm repository in your local set up in the following command add Bitnami repo at:

https://charts.bitnami.com/bitnami

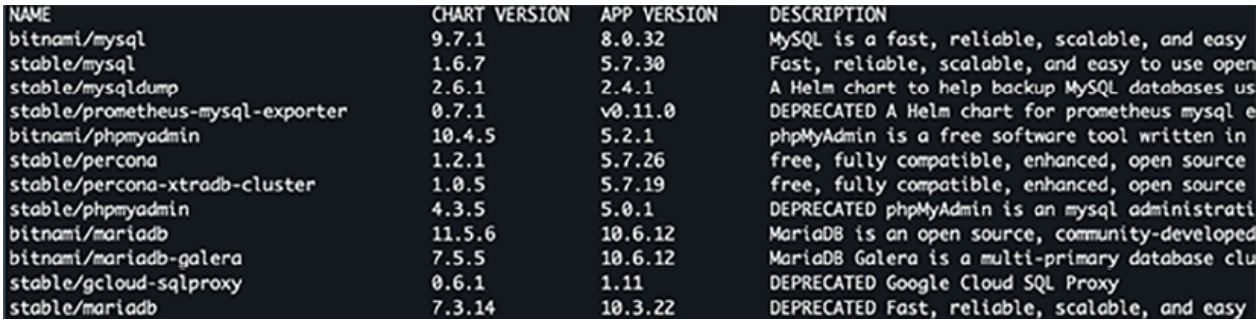
The following command will add a new bitnami chart to the existing repository:

$ Helm repo adds bitnami https://charts.bitnami.com/bitnami

Then you can search the repository using the Helm repo command:

$Helm search repo mysql

Refer to the following figure to search for MySQL in Repo:



NAME	CHART VERSION	APP VERSION	DESCRIPTION
bitnami/mysql	9.7.1	8.0.32	MySQL is a fast, reliable, scalable, and easy
stable/mysql	1.6.7	5.7.30	Fast, reliable, scalable, and easy to use open
stable/mysqldump	2.6.1	2.4.1	A Helm chart to help backup MySQL databases us
stable/prometheus-mysql-exporter	0.7.1	v0.11.0	DEPRECATED A Helm chart for prometheus mysql e
bitnami/phpmyadmin	10.4.5	5.2.1	phpMyAdmin is a free software tool written in
stable/percona	1.2.1	5.7.26	free, fully compatible, enhanced, open source
stable/percona-xtradb-cluster	1.0.5	5.7.19	free, fully compatible, enhanced, open source
stable/phpmyadmin	4.3.5	5.0.1	DEPRECATED phpMyAdmin is an mysql administrati
bitnami/mariadb	11.5.6	10.6.12	MariaDB is an open source, community-developed
bitnami/mariadb-galera	7.5.5	10.6.12	MariaDB Galera is a multi-primary database clu
stable/gcloud-sqlproxy	0.6.1	1.11	DEPRECATED Google Cloud SQL Proxy
stable/mariadb	7.3.14	10.3.22	DEPRECATED Fast, reliable, scalable, and easy

Figure 4.24: Display MySQL in the repo

Execute the Helm repo list command to get the repo list, showing the available repos, including the Bitnami repo:

$helm repo list

Once you find the chart next step is to install the chart bitnami/mysql in your cluster.

If no release name is provided, then the default name is generated. The install command downloads the chart from the repo and installs it locally. Each installation is called a release, and each release has a release name:

$helm install bitnami/mysql --generate-name

Refer to Figure 4.25 and Figure 4.26 as it provides the detail of the release, including the deployment status and other details of MySQL components like login credentials:

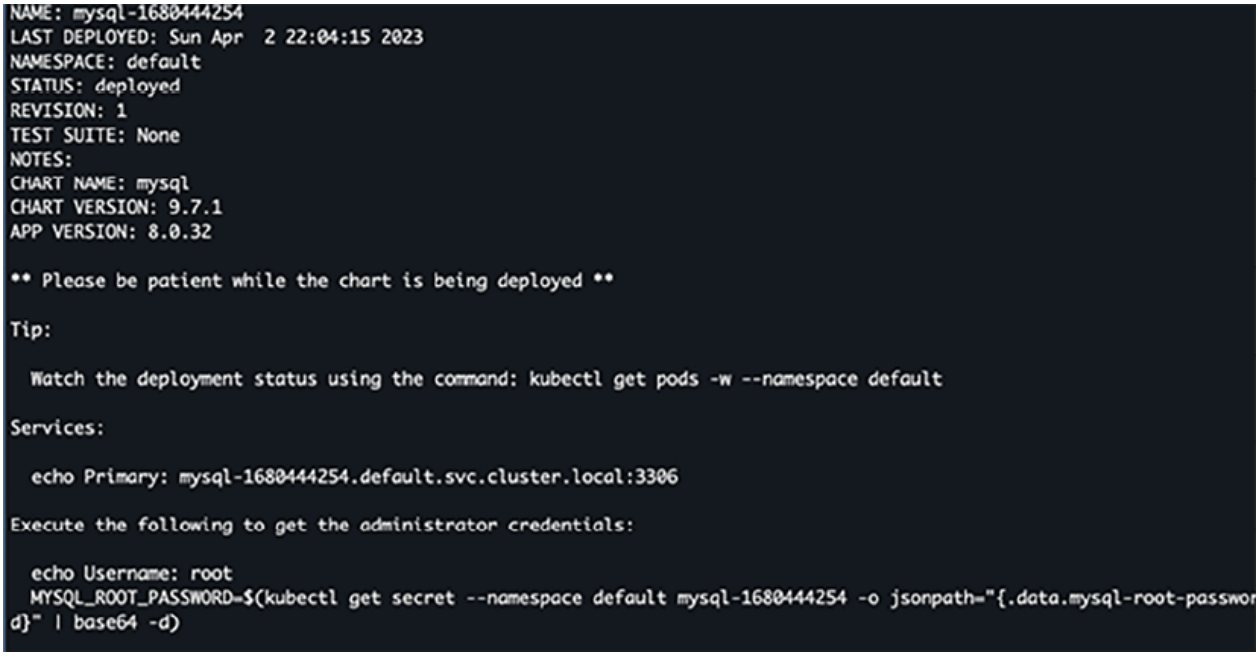


Figure 4.25: Details of MySQL release

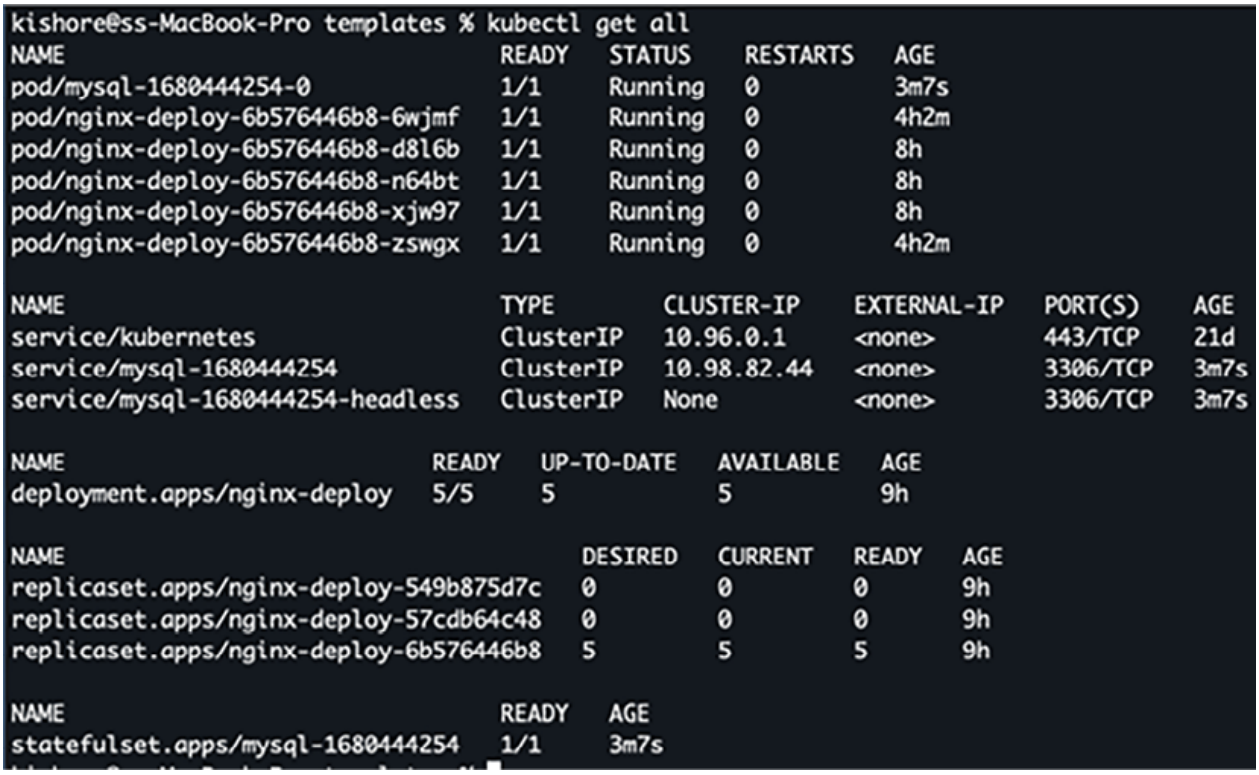

kishore@ss-MacBook-Pro templates % kubectl get all
NAME READY STATUS RESTARTS AGE
pod/mysql-1680444254-0 1/1 Running 0 3m7s
pod/nginx-deploy-6b576446b8-6wjmf 1/1 Running 0 4h2m
pod/nginx-deploy-6b576446b8-d8l6b 1/1 Running 0 8h
pod/nginx-deploy-6b576446b8-n64bt 1/1 Running 0 8h
pod/nginx-deploy-6b576446b8-xjw97 1/1 Running 0 8h
pod/nginx-deploy-6b576446b8-zswgx 1/1 Running 0 4h2m
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 21d
service/mysql-1680444254 ClusterIP 10.98.82.44 <none> 3306/TCP 3m7s
service/mysql-1680444254-headless ClusterIP None <none> 3306/TCP 3m7s
NAME READY UP-TO-DATE AVAILABLE AGE
deployment.apps/nginx-deploy 5/5 5 5 9h
NAME DESIRED CURRENT READY AGE
replicaset.apps/nginx-deploy-549b875d7c 0 0 0 9h
replicaset.apps/nginx-deploy-57cdb64c48 0 0 0 9h
replicaset.apps/nginx-deploy-6b576446b8 5 5 5 9h
NAME READY AGE
statefulset.apps/mysql-1680444254 1/1 3m7s

Figure 4.26: Command displays all Kubernetes objects

At this point, Helm chart work is done, and it has installed MySQL related resources.

If you want to write the contents of the values.yaml file of the bitnami/mysql chart to standard output, execute the following command:

$helm show values bitnami/mysql

Some other Helm commands you should be aware of:

This command lists installed packages:

$ Helm list

Uninstall packages from Helm:

$Helm uninstall < release name>

Use Helm pull command only to download the Helm packages and not install it:

$Helm pull –untar bitnami/mysql

Once downloaded, you can change the values of the chart from values.yaml and then install again using:

Helm install release4 ./mysql

Conclusion

In this chapter, we revisited the deployment object and looked at the various deployment strategies. We also discussed deployment revisions and how to roll back deployment. Lastly, we discussed Helm and how to create Helm from the local and remote chart.

In the next chapter, we will dive deep into Pod design and concepts.

Practice test deployment and Helm

Create a deployment named The deployment should be launched from the nginx image with two replicas. Verify the status of the deployment. As part of rolling update, update the image to Verify the status of the deployment.

Create a deployment named The deployment should have three replicas of nginx image. Once the deployment is created, verify the maxSurge and maxUnavailable parameters. Edit the maxUnavailable to 0 and maxSurge to 30% on the live deployment object. Once those two parameters are modified, change the image of the deployment to Make sure to use the record instruction on rolling updates.

Create a new Deployment named server-deployment. The Deployment should control two replicas using the image grand-server:1.4.6. Inspect the Deployment and find the root cause for its failure. Fix the issue by assigning the image nginx instead. Inspect the rollout history. How many revisions would you expect to see?

Using Helm from Bitnami repo, install WordPress

Give the release name as release1.

List the Helm package.

Uninstall the Helm package.

Solutions

Solution for question 1:

Step 1: Create a Deployment; using the below YAML file, create a deployment declarative.

```
apiVersion: apps/v1

kind: Deployment

metadata:

labels:

  app: nginx

name: deploy-test

spec:

replicas: 2

selector:
```

```
  matchLabels:
    app: nginx
template:
  metadata:
    labels:
      app: nginx

  spec:
    containers:
    - image: nginx
      name: container-test
```

Step 2: Update the image

```
kubectl set image deployment deploy-test container-test=nginx:alpine --record=true
```

Step 3: Verify the status of deployment: This can be verified in multiple ways, here are some of the approaches:

kubectl rollout status deployment deploy-test

Solution for question 2:

Step 1: Create the deployment. Using the below YAML file, create deployment declarative:

apiVersion: apps/v1

kind: Deployment

metadata:

labels:

  run: deploy-config

name: deploy-config

spec:

replicas: 3

selector:

  matchLabels:

```
    run: deploy-config
template:
  metadata:
    labels:
      run: deploy-config
  spec:
    containers:
    - image: nginx
      name: deploy-config
```

Step 2: Verification of the max surge and max unavailable: To verify the maxSurge and maxUnavailable parameters, run the following command:

```
$kubectl get deployment deploy-config -o yaml
```

Step 3: Modify the max surge and max unavailable parameters:

```
$kubectl edit deployment deploy-config
```

Edit the maxUnavailable to 0 and maxSurge to 30% and save.

Set 4: Set a new image to the deployment and record the changes;

kubectl set image deployment deploy-config deploy-config=nginx:alpine —record=true

Solution for question 3:

Step 1: Create a Deployment using imperative way

$ kubectl create deployment server-deployment --image=grand-server:1.4.6

Step 2: Scale replicas

kubectl scale deployment server-deployment --replicas=2

The problem is that the assigned image does not exist. Having a look at one of its Pods will reveal the issue in the events log:

$kubectl get deployments

$kubectl get Pods

Step 3: set the image nginx

$ kubectl set image deployment server-deployment grand-server=nginx

Step 4: rollout history

Kubectl rollout history deployment server-deployment

Solution for question 4:

Step 1: Install wordpress with release name release 1:

$ helm install bitnami/wordpress –release1

Step 2: List the help packages:

$helm list

Step 3: Uninstall release 1:

$Helm uninstall release1

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 5

Pod Design and Concepts

Introduction

In this chapter, we will dive deeply into Pod, and learn labels and annotations. We will also investigate Job and CronJobs. And we will end the chapter with some exercises. So, let us get started.

Structure

In this chapter, we will cover the following topics:

Review labels, label selections

Annotations

Job

CronJob

Objectives

The objective of this chapter is to ensure you have a good understanding of Kubernetes concepts like labels and annotations. and how Job and CronJob are used in the Kubernetes world.

Review labels and labels selections

In Chapter Introduction to Kubernetes, Docker, and minikube, we saw how Kubernetes is well-suited for microservice architecture. In a microservice-based architecture, the number of Pods running each service can easily exceed 100. We have seen some complex large Kubernetes microservice applications running around 500 Kubernetes objects in a Kubernetes cluster. Microservice application consists of various components: front-end, back-end, and database. These components run on Pods, leading to hundreds of Pods in the Kubernetes cluster. This brings a new challenge in organizing these Pods based on their functionality. Let us say you want to apply certain Kubernetes commands to Pods running the front end of the application; how to differentiate or carve out only those Pods running the front-end application from a cluster of 100 Pods.

This is where labels become handy. In short, labels help you to organize Kubernetes objects.

Let us try to understand this with an example:

Say there is a blog application you are developing which is running on a Kubernetes cluster. The application comprises various front-end modules, back-end services, and database and API call-out modules. All these modules are running on multiple Pods. The following figure shows that Pods is running multiple microservices with multiple replicas. As you can see, the structure is uncategorized, and it is difficult to list out only Pods running UI or managed service:

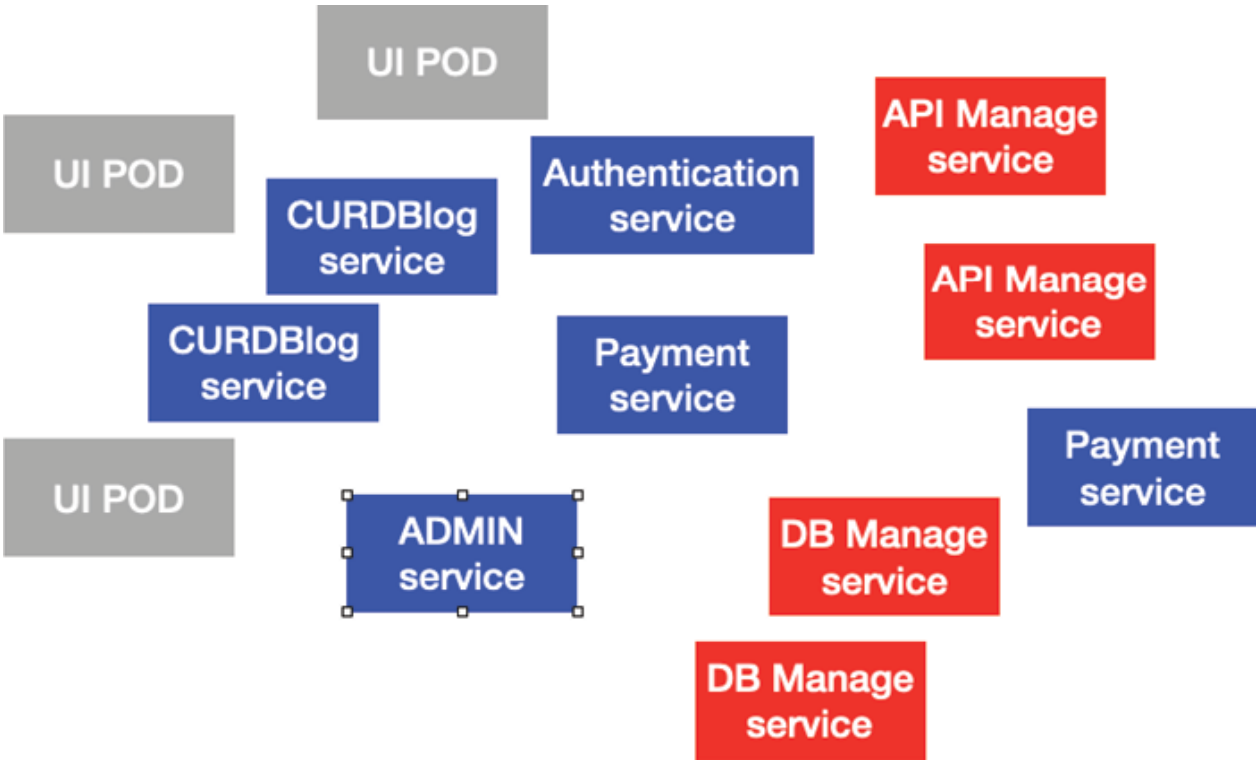


Figure 5.1: Kubernetes objects not categorized

Labels help to organize Kubernetes objects. The label is a key-value pair that you attach to the Kubernetes resource. It helps you understand how the various Kubernetes objects are structured and their hierarchy and enables you to filter kubectl outputs.

As seen in the preceding example, if we structure the objects using labels and add labels to them, the cluster will be more organized, as shown below:

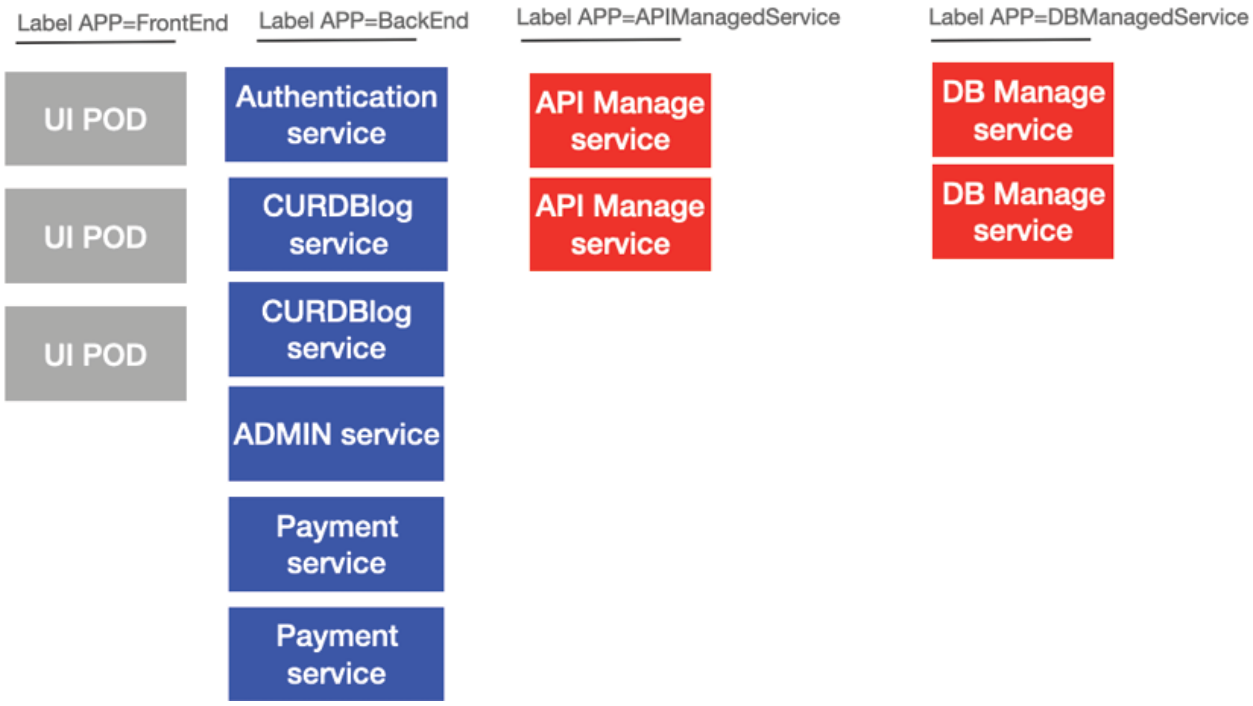


Figure 5.2: Kubernetes categorized using label

Labels use key-value pairs, so in the above example, all the UI Kubernetes objects are tagged as back-end service as and so on.

Labels can be created in an imperative or declarative way. Let us see how a label can be added to a Pod using these two ways.

As shown below, for the imperative approach, you can create the label using -l while creating the object, -l is the short form for the label:

$kubectl run nginx --image=nginx -l environment=frontend,tier=ui

The above command will create a Pod named nginx and attach two labels environment=frontend, to it.

As the object has label, you can selectively filter kubectl outputs to the objects you need with -l command, as shown below:

$ kubectl get po -l environment=frontend

The above command selectively filters the Pods with label Note though the Pod is attached to two label, you can still selectively filter the Pod based on one label.

Refer to the following Figure

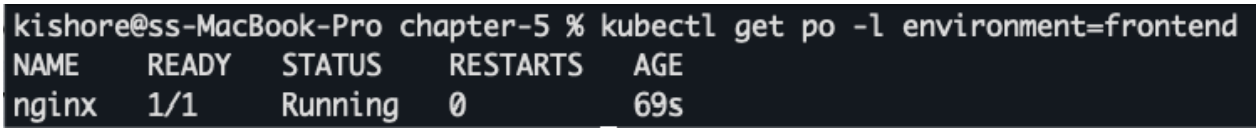

kishore@ss-MacBook-Pro chapter-5 % kubectl get po -l environment=frontend
NAME READY STATUS RESTARTS AGE
nginx 1/1 Running 0 69s

Figure 5.3: Using label to selectively filter the Pod

The next approach is to create the label declarative, as shown below, the label is part of metadata section:

apiVersion: v1

kind: Pod

metadata:

name: nginx

labels:

environment: frontend

tier: ui

spec:

containers:

- name: nginx

image: nginx

The show label command is used to show the labels of the Kubernetes resources.

$kubectl get po --show-labels

The above command shows the Pods with attached labels in a Kubernetes cluster.

Refer to the following Figure

```
kishore@ss-MacBook-Pro chapter-5 % kubectl get po --show-labels
NAME        READY   STATUS    RESTARTS   AGE   LABELS
backend     1/1     Running   0          34s   env=prod,tier=be
datalayer   1/1     Running   0          14s   env=qa,tier=data
frontend    1/1     Running   0          49s   env=prod,tier=fe
```

Figure 5.4: Using show label to display attached Pod

We had seen earlier that you can select objects by label using the -l is the short form notation of label. It can be used to query both equality-based requirements and set-based requirements. Equality-based requirement can use the operators != and set-based requirement is based on a set of values using the operators and

Let us see with an example:

In the Kubernetes cluster, three Pod are running with the below labels, as shown in Figure

```
kishore@ss-MacBook-Pro chapter-5 % kubectl get po --show-labels
NAME        READY   STATUS    RESTARTS   AGE   LABELS
backend     1/1     Running   0          34s   env=prod,tier=be
datalayer   1/1     Running   0          14s   env=qa,tier=data
frontend    1/1     Running   0          49s   env=prod,tier=fe
```

Figure 5.5: Pods with attached labels in the cluster

$ kubectl get po -l env!=prod

The above equality-based command will only filter out Pods which does not have the label and that is, only one Pod with the name as for this Pod, the env value is qa (as shown in Figure

```
kishore@ss-MacBook-Pro chapter-5 % kubectl get po  -l env!=prod
NAME        READY   STATUS    RESTARTS   AGE
datalayer   1/1     Running   0          11m
```

Figure 5.6: Filter out Pods based on equality

$ kubectl get po -l ‘env in (prod)’

The above set-based requirement will only get Pods that have the matching value of env specified, as shown in Figure

```
kishore@ss-MacBook-Pro chapter-5 % kubectl get po  -l 'env in (prod)'
NAME       READY   STATUS    RESTARTS   AGE
backend    1/1     Running   0          15m
frontend   1/1     Running   0          16m
```

Figure 5.7 Filter out Pods based on set

Label at any point in time can be added, removed, or modified.

$ kubectl label pod labelpod env=prod

The above command adds label to the existing set of as shown in Figure

```
kishore@ss-MacBook-Pro chapter-5 % kubectl get po labelpod --show-labels
NAME       READY   STATUS    RESTARTS   AGE   LABELS
labelpod   1/1     Running   0          96s   env=prod,tier=fe
```

Figure 5.8: Added label to the Pod

$ kubectl label pod labelpod env=qa --overwrite

The above commands modify the label value, as shown in Figure the value of env is updated to

```
kishore@ss-MacBook-Pro chapter-5 % kubectl get po labelpod --show-labels
NAME       READY   STATUS    RESTARTS   AGE     LABELS
labelpod   1/1     Running   0          5m38s   env=qa,tier=fe
```

Figure 5.9: Updated label to the Pod

$kubectl label pod labelpod env-

The above command removes the label from the Pod.

As shown in Figure the env label is no longer associated with the Pod:

```
kishore@ss-MacBook-Pro chapter-5 % kubectl get po labelpod --show-labels
NAME       READY   STATUS    RESTARTS   AGE     LABELS
labelpod   1/1     Running   0          7m55s   tier=fe
```

Figure 5.10: Removed label from the Pod

Thus, you have seen how Labels in Kubernetes are an essential tool for querying, filtering, and sorting Kubernetes objects.

Annotations

We saw in earlier sections that labels are key-value pairs attached to Kubernetes objects. Labels identify attributes of Kubernetes objects that bring some meaning and are relevant to the user. There could be instances when you need to attach non-identifying informative information to the Kubernetes objects. And this is where you use annotation.

Annotations are used to attach any metadata information to Kubernetes objects. This is used if you want to provide metadata information of the Kubernetes objects that is relevant and helpful to fellow project team members. For example, if you want to provide additional comments to the Kubernetes object, such as who in the dev team is the author of the application running on the Pod, why the Pod was created, and which branch from the bit bucket the application refers to, you can use annotation.

Refer to the following example:

```
apiVersion: v1

kind: Pod

metadata:

name: nginx

annotations:
```

```
  author: “@kishore”

  origin: “The Pod will run the front-end application”


  branch:  “master”

Labels:

  environment: frontend

  tier: ui

spec:

containers:

- name: nginx

  image: nginx
```

Annotations can be created in an imperative or declarative way. Let us see how annotations can be added to a Pod using these two ways.

As shown below, you can create the annotation using the annotate command for the imperative approach:

```
$kubectl annotate pod backend description=”for be application”
```

Once the Pod is annotated, you can use the describe or get commands to retrieve the assigned annotations.

$kubectl describe pod backend | grep -C2 “application”

The above command displays the annotation of the as shown in Figure

```
kishore@ss-MacBook-Pro ~ % kubectl describe pod  backend | grep -C2 "application"
Labels:           env=prod
                  tier=be
Annotations:      description: for be application
Status:           Running
IP:               10.244.0.101
```

Figure 5.11: Inspect the annotation of Pod

You can use the imperative command to modify or delete the annotation.

To update the annotation, use the below command use - - overwrite to update the annotation.

kubectl annotate pod backend description=”The application resides connect to backend application” --overwrite

The above command edits the annotation of the as shown in Figure

```
kishore@ss-MacBook-Pro ~ % kubectl describe pod  backend | grep -C2 "application"
Labels:           env=prod
                  tier=be
Annotations:      description: The application resides connect to backend  application
Status:           Running
IP:               10.244.0.101
```

Figure 5.12: Inspect the annotation of Pod after edit

You can also delete the annotation using the below command:

$kubectl annotate pod backend description–

The above command removes the annotation from the Pod, here, description is the key to annotation.

Annotation can also be added declarative. For this, update the YAML manifest and add the desired annotations under metadata. Annotation as shown below:

apiVersion: v1

kind: Pod

metadata:

name: nginx

annotations:

  author: “@kishore”

  origin: “The Pod will run the front-end application”

```
  branch:  “master”

Labels:

  environment: frontend

  tier: ui

spec:

containers:

- name: nginx

  image: nginx
```

Job

Pods are the essence of Kubernetes. Pods are created to run forever; it is meant for the continuous operation of an application. If Pod goes down, then Kubernetes will try to start it again. But in some scenarios, you want Pod to execute till a certain number of completed cycles is reached. This is where you use Job, it is useful for the specified number of tasks to be executed by Pods. Some examples that are a good fit for the Job include backup Jobs, calculation, and batch processing.

By default, once the job is completed, Kubernetes do not delete Job and its Pods. You need to explicitly delete it or specify an attribute which is a cleanup mechanism to clean up Jobs and their controlled Pods.

Use create job command to create the job from kubectl imperatively:

```
$ kubectl create job count --image=busybox -- /bin/sh -c ‘counter=0; \

  while [ $counter -lt 5 ]; do counter=$((counter+1)); echo “$counter”; \

  sleep 3; done;’
```

The above command will create a job count with an image busy box, which will print values from 1 to 5, and once this job is completed, the job is finished.

Refer to the following Figure

```
kishore@ss-MacBook-Pro ~ %  kubectl create job count --image=busybox -- /bin/sh -c 'counter=0; \
  while [ $counter -lt 5 ]; do counter=$((counter+1)); echo "$counter"; \
  sleep 3; done;'
job.batch/count created
kishore@ss-MacBook-Pro ~ % kubectl get job
NAME    COMPLETIONS   DURATION   AGE
count   0/1           6s         6s
kishore@ss-MacBook-Pro ~ % kubectl get job
NAME    COMPLETIONS   DURATION   AGE
count   0/1           10s        10s
kishore@ss-MacBook-Pro ~ % kubectl get job
NAME    COMPLETIONS   DURATION   AGE
count   1/1           26s        39s
```

Figure 5.13: Create job command

As a part of creating the corresponding Pod is also created, which uses Job name as a prefix in its name, for example, where the count is the name of the job.

Refer to the following Figure

```
kishore@ss-MacBook-Pro ~ % kubectl get po,job
NAME                READY   STATUS      RESTARTS   AGE
pod/count-scqbw     0/1     Completed   0          3m10s

NAME                COMPLETIONS   DURATION   AGE
job.batch/count     1/1           26s        3m10s
```

Figure 5.14: Corresponding Pod for the Job

You can query the corresponding check log to see if the Job is completed.

$kubectl logs count-scqbw

The above command will query the Pod and if the Job of printing count values till 5 got completed.

Refer to the following Figure

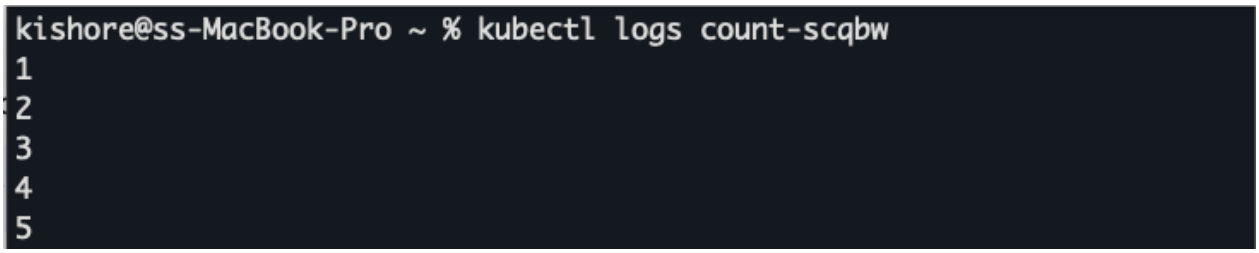


Figure 5.15: Query the logs of the Pod

You can also declaratively create Job using the YAML manifest.

```
apiVersion: batch/v1

kind: Job

metadata:

  name: count

spec:

  template:

    spec:
```

```
    containers:

    - name: counter

      image: busybox

      command:

      - /bin/sh

      - -c

      - counter=0; while [ $counter -lt 5 ]; do counter=$((counter+1)); \

        echo “$counter”; sleep 3; done;

  restart policy: Never
```

The template in the YAML manifest refers to the Pod specification of the Job.

By default, the Job completes after one run, and there is only one Pod which executes. The Pods, attribute completions and parallelism define this.

If you grep the count job, you will see the completions and parallelism set to 1

Refer to the following Figure

```
kishore@ss-MacBook-Pro ~ % kubectl get jobs count -o yaml | grep -C 1 "completions"
  completionMode: NonIndexed
  completions: 1
  parallelism: 1
```

Figure 5.16: Compilation and parallelism count of the Pod

The parameter values of completions and parallelism can be changed.

Refer to the following table to understand the significance of each value:

Table 5.1: Significance of compilation and parallelism values

Cron job

Linux has a cron utility which is a scheduling utility, Cronjob is the scheduling utility of job in the Kubernetes world.

In the previous section, we have seen job, it is used to run a task a specific number of times till completion. CronJobs are used for tasks that need to run regularly at a specific time based on a schedule.

Like all other Kubernetes object, CronJob can be created imperative

kubectl create cronjob current-date --schedule=”* * * * *” --image=busybox -- /bin/sh -c ‘echo “Current date: $(date)”’

The preceding command has schedule set to “* * * * This will schedule the job to run every minute.

$kubectl get cronjobs

The above command will list down the

Refer to the following Figure

```
kishore@ss-MacBook-Pro ~ % kubectl get cronjobs
NAME           SCHEDULE    SUSPEND   ACTIVE   LAST SCHEDULE   AGE
current-date   * * * * *   False     0        18s             64s
kishore@ss-MacBook-Pro ~ % kubectl get cronjobs
NAME           SCHEDULE    SUSPEND   ACTIVE   LAST SCHEDULE   AGE
current-date   * * * * *   False     0        30s             76s
kishore@ss-MacBook-Pro ~ % kubectl get cronjobs
NAME           SCHEDULE    SUSPEND   ACTIVE   LAST SCHEDULE   AGE
current-date   * * * * *   False     0        50s             96s
```

Figure 5.17: Cron job listing

You can refer to the managed Pods of cronjobs by executing the get Pods command. Like jobs, you can identify them by the name prefix.

Refer to the following Figure

```
kishore@ss-MacBook-Pro ~ % kubectl get pods
NAME                             READY   STATUS      RESTARTS   AGE
current-date-28027223-w69w6      0/1     Completed   0          2m5s
current-date-28027224-964dt      0/1     Completed   0          65s
current-date-28027225-wjw9f      0/1     Completed   0          5s
```

Figure 5.18: Pods corresponding to CronJobs

You can query the corresponding Pod and check its log to see if the Job is completed.

$kubectl logs current-date-28027223-w69w6

The above command will query the Pod and if the Job of printing date is completed.

Refer to the following Figure

```
kishore@ss-MacBook-Pro ~ % kubectl logs current-date-28027224-964dt
Current date: Sun Apr 16 08:24:03 UTC 2023
kishore@ss-MacBook-Pro ~ % kubectl logs current-date-28027223-w69w6
Current date: Sun Apr 16 08:23:04 UTC 2023
```

Figure 5.19: Query the logs of the Pod

Like Job, CronJob can also be created YAML manifest, that is, declarative way.

```
apiVersion: batch/v1beta1

kind: CronJob

metadata:

  name: current-date



spec:

  schedule: “* * * * *”

  jobTemplate:

    spec:

      template:
```

```
spec:
  containers:
  - name: current-date
    image: busybox
    args:
    - /bin/sh
    - -c
    - 'echo "Current date: $(date)"'
  restartPolicy: OnFailure
```

CronJob also retains the successful jobs and failure jobs. By default, it retains the last 3 successful and failure jobs. The success and failure are captured via successfulJobsHistoryLimit and failedJobsHistoryLimit

```
$kubectl get cronjobs current-date -o yaml | grep successfulJobsHistoryLimit
```

The above commands list out the successful job history of the cron job.

Refer to the following Figure

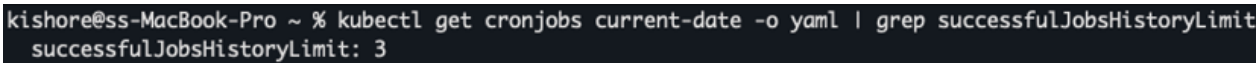

kishore@ss-MacBook-Pro ~ % kubectl get cronjobs current-date -o yaml | grep successfulJobsHistoryLimit
successfulJobsHistoryLimit: 3

Figure 5.20: Grep the successful job history

Conclusion

In this chapter, we looked at the various Pod design and concepts and discussed how jobs and CronJobs are used. In the next chapter, we will look into multi-container Pod, its need, use cases, and design patterns.

Practice test

Create three Pods that use the image nginx. The names of the Pods should be and Assign the label tier=frontend to pod-1 and the label tier=backend to pod-2 and

Continuing with the previous question, assign the annotation with the branch equal to the master value to pod-1 and

Continuing with the previous question, Assign all Pods the label team=avengers

Continuing with the previous question, use label selection to find all Pods with the team avengers, which are considered a backend service.

Create a new CronJob named search with image nginx. the Job should run a curl command for The execution should occur every minute.

Solution

Solution for question 1:

Start by creating the Pod imperative:

$kubectl run pod1 --image=nginx -l tier=frontend

$kubectl run pod2 --image=nginx -l tier=backend

$kubectl run pod3 --image=nginx -l tier=backend

Validate the labels of the Pods

$kubectl get pods --show-labels

Solution for question 2:

Use annotate command to annotate the Pods imperatively:

$kubectl annotate pod pod1 pod3 branch=”master”

Solution for question 3:

Use label command to assign a new label to the Pods imperatively:

$kubectl label pod pod1 pod2 pod3 team=”avengers”

Solution for question 4:

Use the equality-based and set-based filter criteria:

$ kubectl get pods -l tier=backend,’team in (avengers)’

Solution for question 5:

$kubectl create cronjob search --schedule=”* * * * *” --image=nginx --
/bin/sh -c ‘curl google.com’

The above command creates a cronjob which pings www.google.com every minute.

You can inspect the cronjob with the following command:

$kubectl get cronjob

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 6

Multi-container Pods and Design Patterns

Introduction

In this chapter, we will dive deeply into multi-containers Pods, its need, use cases, and design patterns. We will end the chapter with some exercises. So, let us get started.

Structure

In this chapter, we will cover the following topics.

Multi-container Pod

Use cases

Sidecar pattern

Ambassador pattern

Adapter pattern

Init container

Objectives

The objective of this chapter is to ensure you have a good understanding of multi container Pods, various design patterns, and related use cases and scenarios where it is used.

Multi-container Pod

In Kubernetes, a Pod is the smallest deployable unit that can run one or more containers. Till now, we have seen running one container in a Pod. For most cases, that works, but there are some cases in which you want to go for the multi-container Pod anyway.

A multi-container Pod includes multiple containers. So, when multiple containers are running in a Pod, it is obvious that all these containers will be sharing the same namespace and can communicate via localhost. The multiple containers in the Pod can share the same shared volumes. A good use case for multi-container Pod is when you need to manage tightly coupled containers hosting applications as a single entity. Before we dive into multi-container Pod details, remember that the one-container Pod is standard, and single-container Pods are easier to build and maintain.

Refer to the following Figure which shows multiple containers running in a Pod:

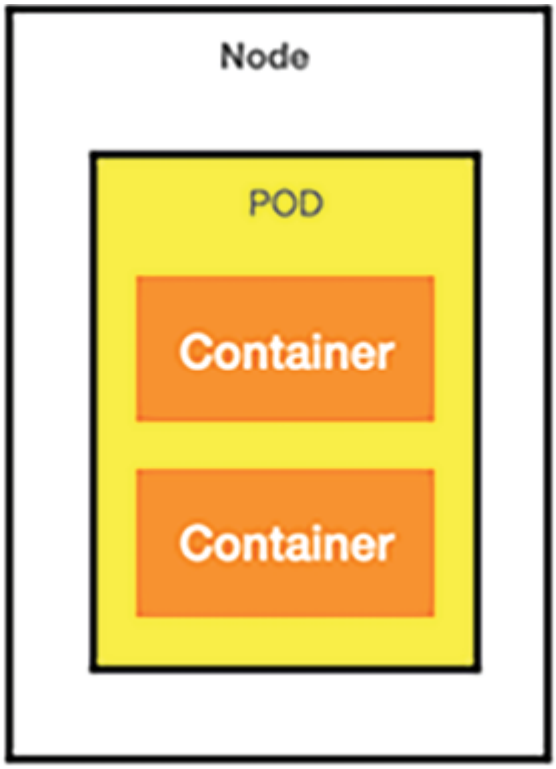


Figure 6.1: Multiple containers running in a Pod

Use cases

Let us see in which scenarios and use cases multi-containers can be helpful.

Multi-container Pods are useful when you need to deploy multiple containers that work together to provide specific functionality, such as a sidecar container that logs or monitors the main container or a container that generates configuration files for the main container. In these cases, the additional containers run to assist the main container and act as a helper function that helps the function deployed in the primary container. Various multi-container patterns can be used for additional functionalities, which we will look at in the next section.

Sidecar pattern

Sidecar, as the name suggested, is a pattern where a secondary container (also known as sidecar) is deployed alongside the main container in the same Pod to enhance or extend the functionalities. A sidecar container enhances or extends the core container functionalities without changing them.

A sidecar pattern provides a helper function to the main container. A common use case of side car pattern is where the main container running the primary application of the Pod generates the log, the side car pattern monitors these logs and if it identifies any error keyword, it notifies the administrators of the application.

Refer to the following Figure which shows the sidecar container pattern:

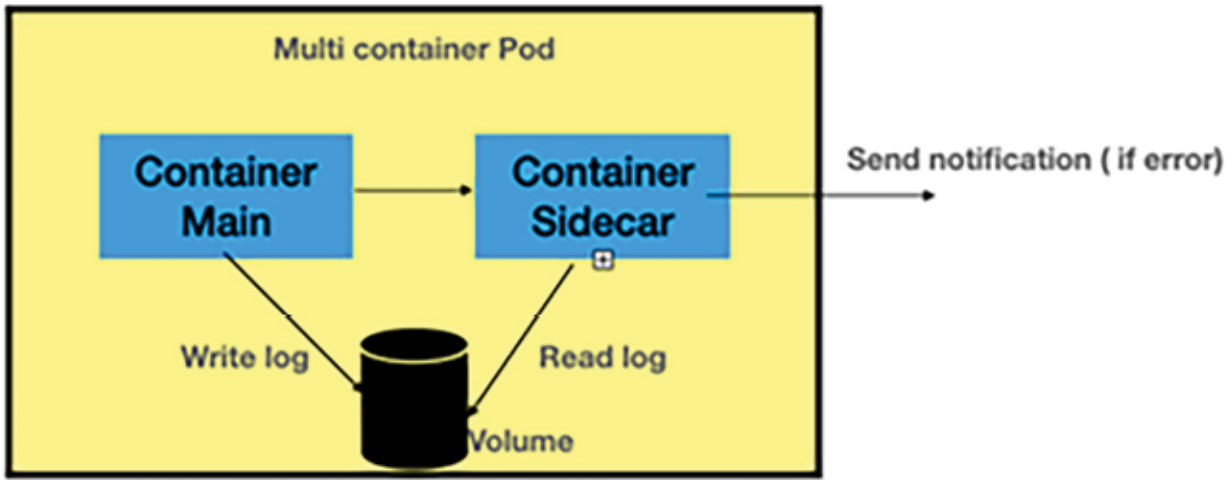


Figure 6.2: Sidecar pattern

The following are some examples where sidecar patterns can be used:

The sidecar container can collect and forward logs generated by the main container to a centralized logging system.

The sidecar can provide metrics to the monitoring system by monitoring the performance and health of the main container

Authentication and The sidecar container can provide an additional layer of security by handling authentication and authorization for the main container.

Load Sidecar can act as a load balancer by managing the load for incoming requests to the main container.

Using sidecar, we separate the concerns and ensure that each container in the Pod has a clear and specific responsibility. This helps in managing complex applications.

Let us look at an example now. The YAML manifest shown below sets up the sidecar pattern. There are two containers defined one named app, which generates the log and another one which reads the generated

kind: Pod

apiVersion: v1

metadata:

  name: sidecarpattern

```
spec:


  volumes:

  - name: log-vol

    emptyDir: {}

  containers:

  - name: app

    image: busybox

    command: [“/bin/sh”]

    args: [“-c”, “while true; do date >> /var/log/nginx/date.txt; sleep 10;done”]

    volumeMounts:

    - name: log-vol

      mountPath: /var/log/nginx

  - name: sidecar

    image: nginx
```

```
volumeMounts:

- name: log-vol

  mountPath: /var/log/
```

Once you create the Pod using the above manifest file, you will notice that the overall number of containers will show 2 and the status of these Pods are running.

Refer to Figure which shows the overall running containers in Pod as 2:

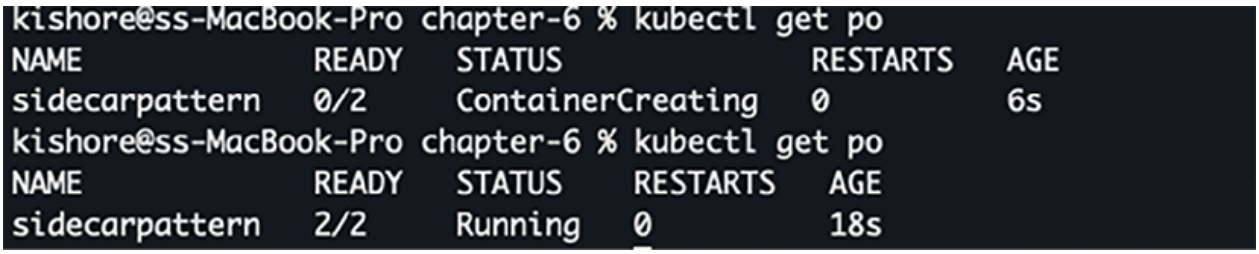

kishore@ss-MacBook-Pro chapter-6 % kubectl get po
NAME READY STATUS RESTARTS AGE
sidecarpattern 0/2 ContainerCreating 0 6s
kishore@ss-MacBook-Pro chapter-6 % kubectl get po
NAME READY STATUS RESTARTS AGE
sidecarpattern 2/2 Running 0 18s

Figure 6.3: Sidecar pattern

If you query the containers, you will see both containers have access to logs which generate date.

$kubectl exec -it sidecarpattern -c app /bin/sh

The above command will query the main container.

Refer to Figure which shows the query to the main container:

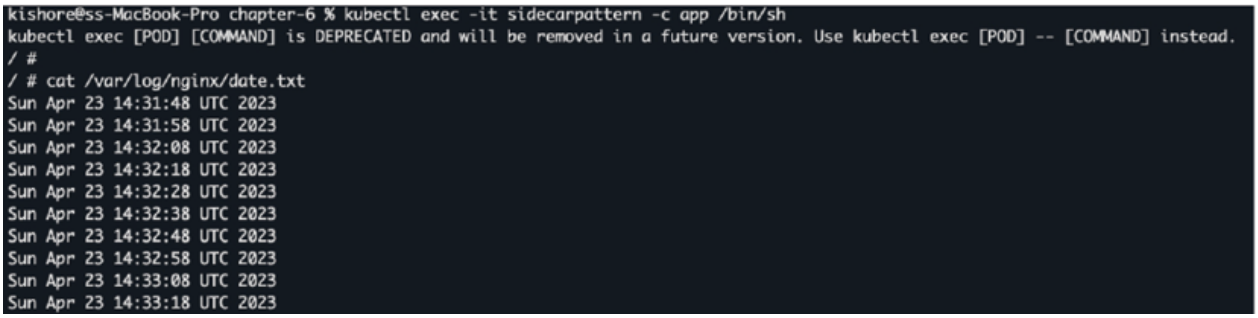

kishore@ss-MacBook-Pro chapter-6 % kubectl exec -it sidecarpattern -c app /bin/sh
kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in a future version. Use kubectl exec [POD] -- [COMMAND] instead.
/ #
/ # cat /var/log/nginx/date.txt
Sun Apr 23 14:31:48 UTC 2023
Sun Apr 23 14:31:58 UTC 2023
Sun Apr 23 14:32:08 UTC 2023
Sun Apr 23 14:32:18 UTC 2023
Sun Apr 23 14:32:28 UTC 2023
Sun Apr 23 14:32:38 UTC 2023
Sun Apr 23 14:32:48 UTC 2023
Sun Apr 23 14:32:58 UTC 2023
Sun Apr 23 14:33:08 UTC 2023
Sun Apr 23 14:33:18 UTC 2023

Figure 6.4: query to the main container

$kubectl exec -it sidecarpattern -c sidecar /bin/sh

The above command will query the sidecar container.

Refer to Figure which shows a query to the sidecar container:

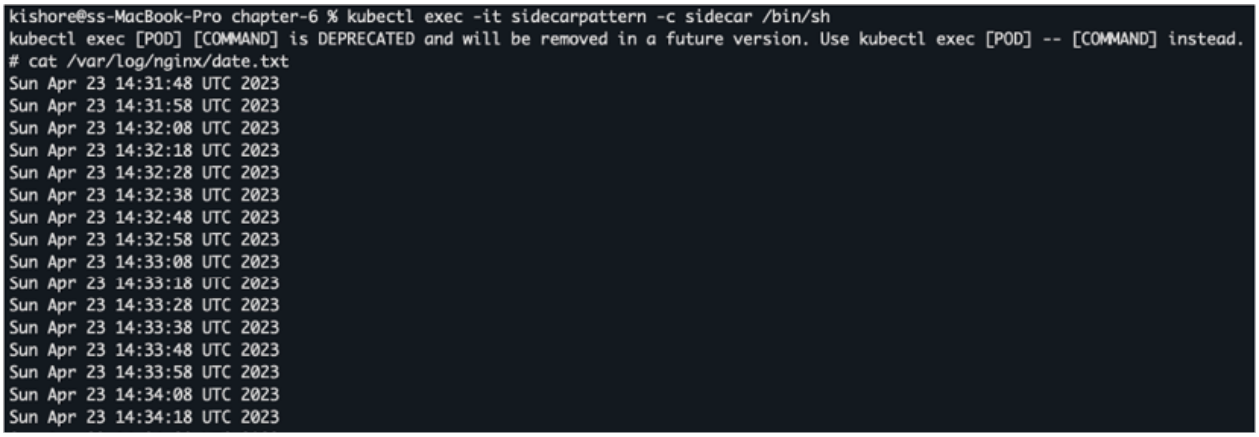

kishore@ss-MacBook-Pro chapter-6 % kubectl exec -it sidecarpattern -c sidecar /bin/sh
kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in a future version. Use kubectl exec [POD] -- [COMMAND] instead.
cat /var/log/nginx/date.txt
Sun Apr 23 14:31:48 UTC 2023
Sun Apr 23 14:31:58 UTC 2023
Sun Apr 23 14:32:08 UTC 2023
Sun Apr 23 14:32:18 UTC 2023
Sun Apr 23 14:32:28 UTC 2023
Sun Apr 23 14:32:38 UTC 2023
Sun Apr 23 14:32:48 UTC 2023
Sun Apr 23 14:32:58 UTC 2023
Sun Apr 23 14:33:08 UTC 2023
Sun Apr 23 14:33:18 UTC 2023
Sun Apr 23 14:33:28 UTC 2023
Sun Apr 23 14:33:38 UTC 2023
Sun Apr 23 14:33:48 UTC 2023
Sun Apr 23 14:33:58 UTC 2023
Sun Apr 23 14:34:08 UTC 2023
Sun Apr 23 14:34:18 UTC 2023

Figure 6.5: Query to the sidecar container

Ambassador pattern

The ambassador pattern is best suited in scenarios where you try to simplify the process of the application needing to access outside service. Let us look at it with an example, say you have an application running on a container that needs to access external service via secured protocol HTTPS. So now, instead of the application managing the secured access to external service, the task of secure communication to external service is delegated to the ambassador container running alongside the primary container. The ambassador container is responsible for secure connectivity to external services.

Refer to Figure which explains the flow:

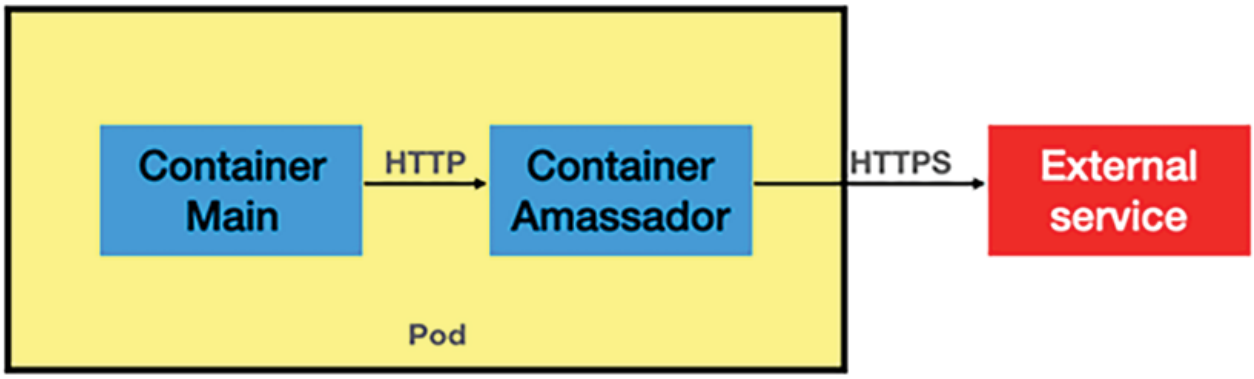


Figure 6.6: Ambassador pattern

Adapter pattern

The adapter pattern, as the name suggests, acts as a bridge between the main application container and other supporting containers. The adapter container runs in parallel with the main container and is typically used when your main container needs to standardize the output format for the external system.

For example, say you have a microservice application running with different tech stacks Java and Python. Each of these microservice applications will output logs in its language-specific format. All the logs across different applications collected are aggregated at the same end point and you want to standardize the logs output. In this case, let the adapter container run across each pod and standardize the log output as needed.

Refer to the following Figure which explains the flow:

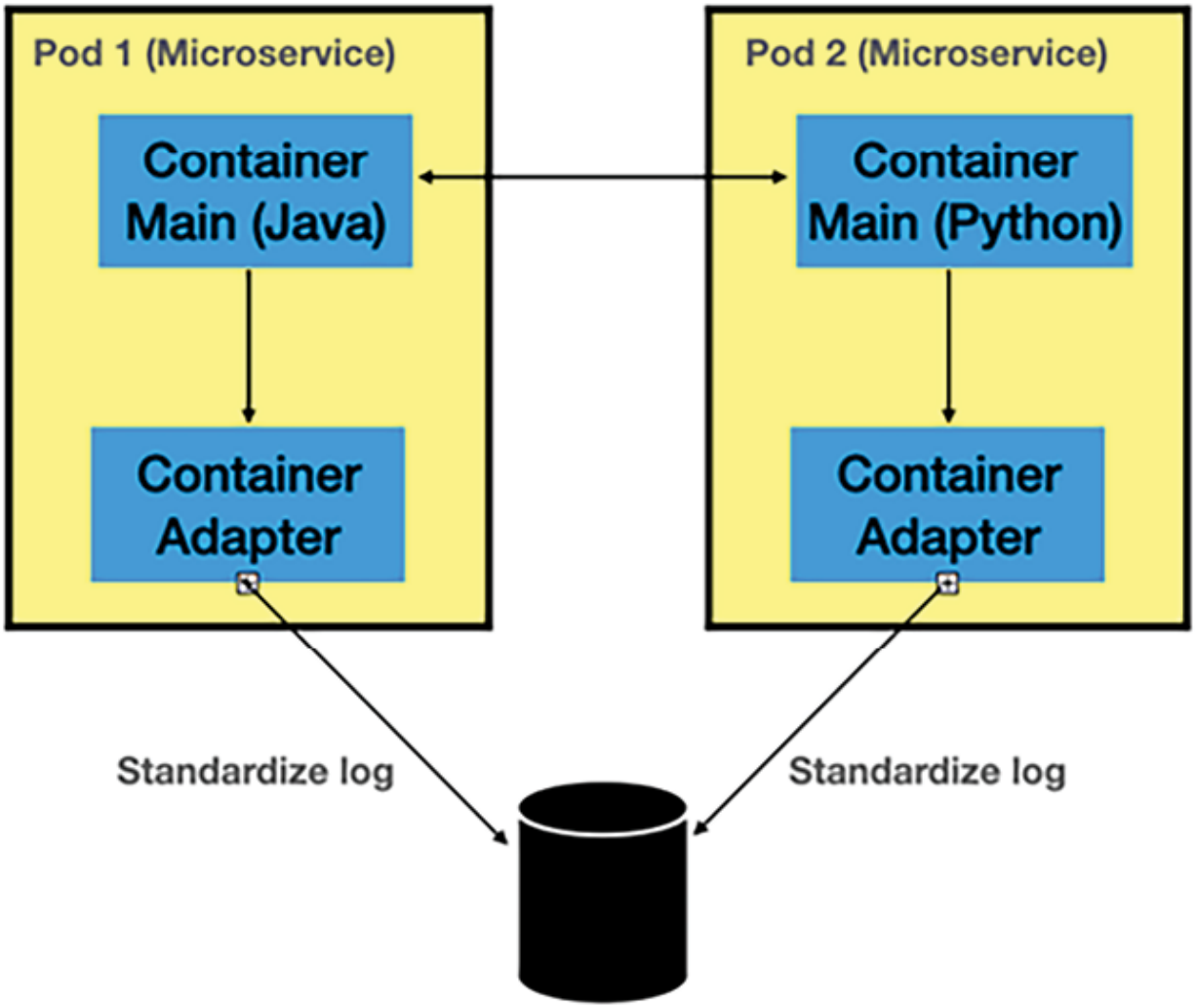


Figure 6.7: Adapter pattern

Let us look at an example now. The YAML manifest shown below sets up the adapter pattern. For example, let us look at a scenario where the application outputs logs in a particular format, then the adapter container changes the output to something standard:

apiVersion: v1

kind: Pod

metadata:

```
  name: adapterpod
spec:

  volumes:
  - name: logs
    emptyDir: {}
  containers:
  - name: app
    image: alpine
    command: [“/bin/sh”]
    args: [“-c”, “while true; do date >> /var/log/app.log; sleep 2;done”]
    volumeMounts:
    - name: logs
      mountPath: /var/log
  - name: adapter
```

```
    image: alpine

    command: [“/bin/sh”]

args: [“-c”, “tail -f /var/log/app.log|sed -e ‘s/^/DateChanged /’ >
/var/log/out.log”]

    volumeMounts:

    - name: logs


      mountPath: /var/log
```

In the manifest, we have main container named app that outputs a stream of dates to a log file. The adapter container named adapter container adds a DateChanged prefix in front.

Once you apply the manifest YAML, both containers are running in the Pod.

Refer to the following Figure which shows Pods running with adapter pattern:

Figure 6.8: Pods running with adapter pattern

$kubectl exec -it adapterpod -c app /bin/sh

The above command will query the main container.

Refer to the following Figure which shows a query to the main container:

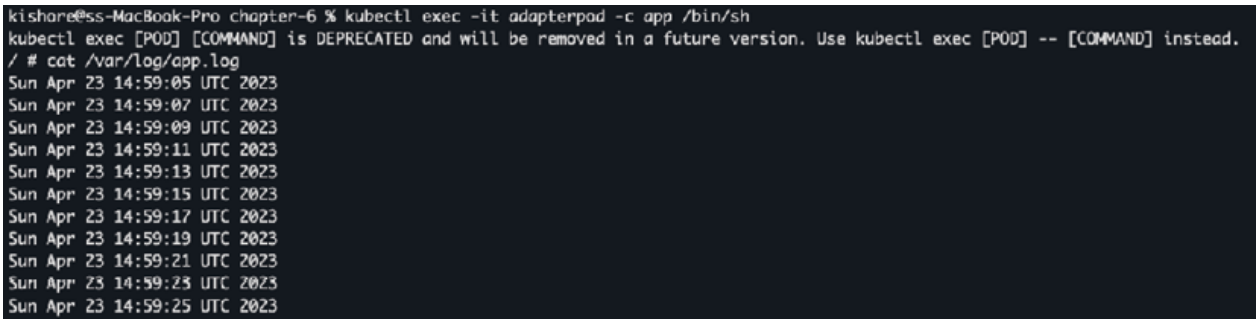

kishore@ss-MacBook-Pro chapter-6 % kubectl exec -it adapterpod -c app /bin/sh
kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in a future version. Use kubectl exec [POD] -- [COMMAND] instead.
/ # cat /var/log/app.log
Sun Apr 23 14:59:05 UTC 2023
Sun Apr 23 14:59:07 UTC 2023
Sun Apr 23 14:59:09 UTC 2023
Sun Apr 23 14:59:11 UTC 2023
Sun Apr 23 14:59:13 UTC 2023
Sun Apr 23 14:59:15 UTC 2023
Sun Apr 23 14:59:17 UTC 2023
Sun Apr 23 14:59:19 UTC 2023
Sun Apr 23 14:59:21 UTC 2023
Sun Apr 23 14:59:23 UTC 2023
Sun Apr 23 14:59:25 UTC 2023

Figure 6.9: Query the main container

$kubectl exec -it adapterpod -c adapter /bin/sh

The above command will query the adapter container and notice that the output is appended via DateChanged text.

Refer to the following Figure which shows query to the adapter container:

```
kishore@ss-MacBook-Pro chapter-6 % kubectl exec -it adapterpod -c adapter /bin/sh
kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in a future version.
/ # cat /var/log/out.log
DateChanged Sun Apr 23 14:59:05 UTC 2023
DateChanged Sun Apr 23 14:59:07 UTC 2023
DateChanged Sun Apr 23 14:59:09 UTC 2023
DateChanged Sun Apr 23 14:59:11 UTC 2023
DateChanged Sun Apr 23 14:59:13 UTC 2023
DateChanged Sun Apr 23 14:59:15 UTC 2023
DateChanged Sun Apr 23 14:59:17 UTC 2023
DateChanged Sun Apr 23 14:59:19 UTC 2023
DateChanged Sun Apr 23 14:59:21 UTC 2023
DateChanged Sun Apr 23 14:59:23 UTC 2023
DateChanged Sun Apr 23 14:59:25 UTC 2023
DateChanged Sun Apr 23 14:59:27 UTC 2023
DateChanged Sun Apr 23 14:59:29 UTC 2023
```

Figure 6.10: Query the adapter container

Init container

Init is a short form for Initial, it is a specialized container that runs before the main container in a Pod. Init container aims to perform some setup or initialization tasks before the main container starts running.

Some scenarios where Init container can be used are:

Waiting for a dependent service or resource to become available before starting the main container.

Running a script or command configures the environment before the main container runs.

Setting up authentication or authorization credentials before the main container runs

Multiple Init containers can be defined in a Pod and run in the specified order. Once all Init Containers have been completed successfully, the main container will start running.

Let us look at an example now. The YAML manifest shown below sets up the init container. For example, let us look at the scenario, will start only after a sleep of 45 seconds is executed, after that, any other containers can be started.

apiVersion: v1

```
kind: Pod

metadata:


  name: initpod

spec:

  containers:

  - name: app

    image: nginx

  initContainers:

  - name: delay

    command: [“sleep”, ‘45’]

    image: busybox
```

If you query the Pod as shown below, you will notice that once the init container is started after 45 seconds of sleep, the main container is up and running.

Refer to the following Figure which shows init container in operation:

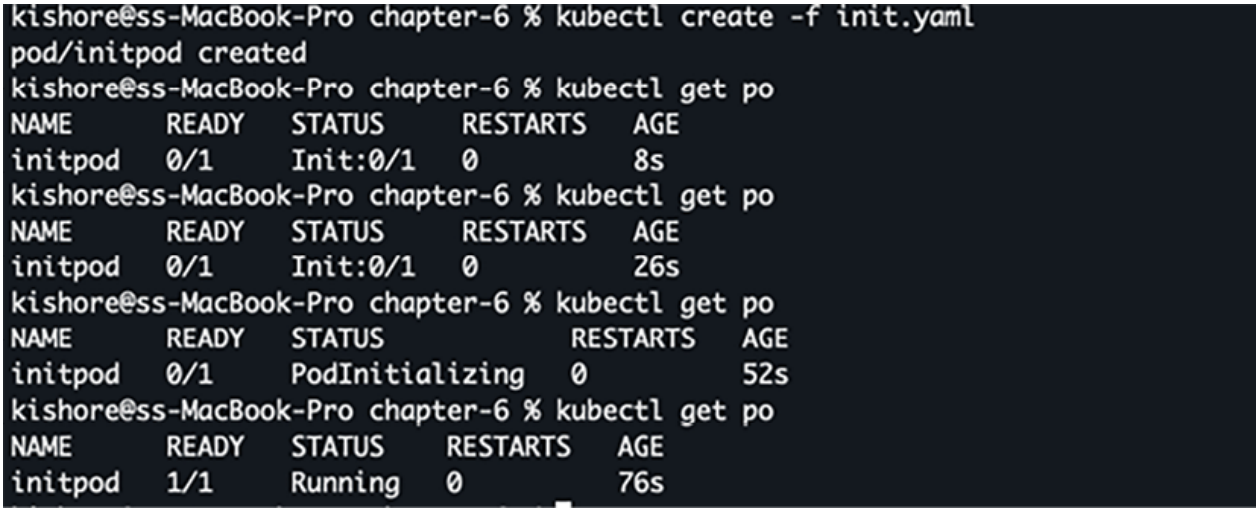

kishore@ss-MacBook-Pro chapter-6 % kubectl create -f init.yaml
pod/initpod created
kishore@ss-MacBook-Pro chapter-6 % kubectl get po
NAME READY STATUS RESTARTS AGE
initpod 0/1 Init:0/1 0 8s
kishore@ss-MacBook-Pro chapter-6 % kubectl get po
NAME READY STATUS RESTARTS AGE
initpod 0/1 Init:0/1 0 26s
kishore@ss-MacBook-Pro chapter-6 % kubectl get po
NAME READY STATUS RESTARTS AGE
initpod 0/1 PodInitializing 0 52s
kishore@ss-MacBook-Pro chapter-6 % kubectl get po
NAME READY STATUS RESTARTS AGE
initpod 1/1 Running 0 76s

Figure 6.11: Init container in operation

Conclusion

In this chapter, we looked at the various multi container Pod design and concepts and discussed how the design of the side car, adapter, and ambassador patterns is designed.

In the next chapter we will look into Kubernetes volumes and persistence storage.

Questions

Create a multi-pod Pod and use a YAML to manifest it. The main application should have a container named mainapp that should use the image nginx. Update the YAML manifest so the Pod defines an init container named initapp that uses the image BusyBox. The init container runs the command wget -O-

Create a multi-container Pod with the name of There should be two containers in the pod. Name of the first container should be first-container, second container should be second container. 1st container should be launched from nginx image, the second container should be launched from BusyBox image. Connect to the first container and run the following command:

apt-get update && apt-get install net-tools.

Connect to the second container and identify the ports in which processes are listening.

Solution

Solution for question 1:

You can start by generating the YAML manifest in dry-run mode. The below command will create a multi-pod.yaml

```
$ kubectl run multi-pod --image= nginx --port=80 --restart=Never \

  -o yaml --dry-run=client > multi-pod.yaml
```

Edit the manifest file by adding the init container and changing some of the default settings that have been generated.

```
apiVersion: v1

kind: Pod

metadata:

  name: multi-pod

spec:

  initContainers:
```

```
  - image: busybox

    name: initapp

    command: [‘sh’, ‘-c’, ‘wget -O- bbc.com’]


  containers:

  - image: nginx

    name: mainapp

    ports:

    - containerPort: 80
```

The above code block will create an init container to the Pod. The init container runs the command wget bbc.com, as mentioned in the question.

Solution for question 2:

Below is Manifest YAML file to create two containers, first container and second container

```
apiVersion: v1
```

```
kind: Pod

metadata:

name: multipod

spec:

containers:

- name: first-container

  image: nginx

- name: second-container


  image: busybox

  command: [“sleep”, “3600”]
```

You can use the kubectl create command to create Pod with two containers.

Next step is to connect to first container and run defined commands, -c is used to connect to a specified container in the Pod

```
$kubectl exec -it multipod -c first-container bash \
```

apt-get update && apt-get install net-tools

Connect to second container and identify ports:

$kubectl exec -it multipod -c second-container sh

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 7

Kubernetes Volumes and Persistence

Introduction

In this chapter, we will dive deeply into Kubernetes volumes, look at different volume types, and discuss how to configure persistent volume and persistence volume claims with Pod. We will end the chapter with some exercises. So, let us get started.

Structure

In this chapter, we will cover the following topics:

Different types of Kubernetes volume types

Pods and deployments volume configuration

Persistent Volume and PersistentVolumeClaim

Configure Pod with PV and PVC

Storage class

Objectives

This chapter aims to ensure you understand Kubernetes volumes, their different types, and how to configure persistent volume and persistence volume claims.

Different types of Kubernetes volume types

The lowest entity in the Kubernetes ecosystem is the container. Each container residing in a Pod has a temporary file system, and applications deployed and running on these containers use these file systems to read and write data. This temporary file system exists till container is alive. Once container is restarted, these temporary file system starts with a clean state.

Refer to the following Figure which shows the temporary file system:

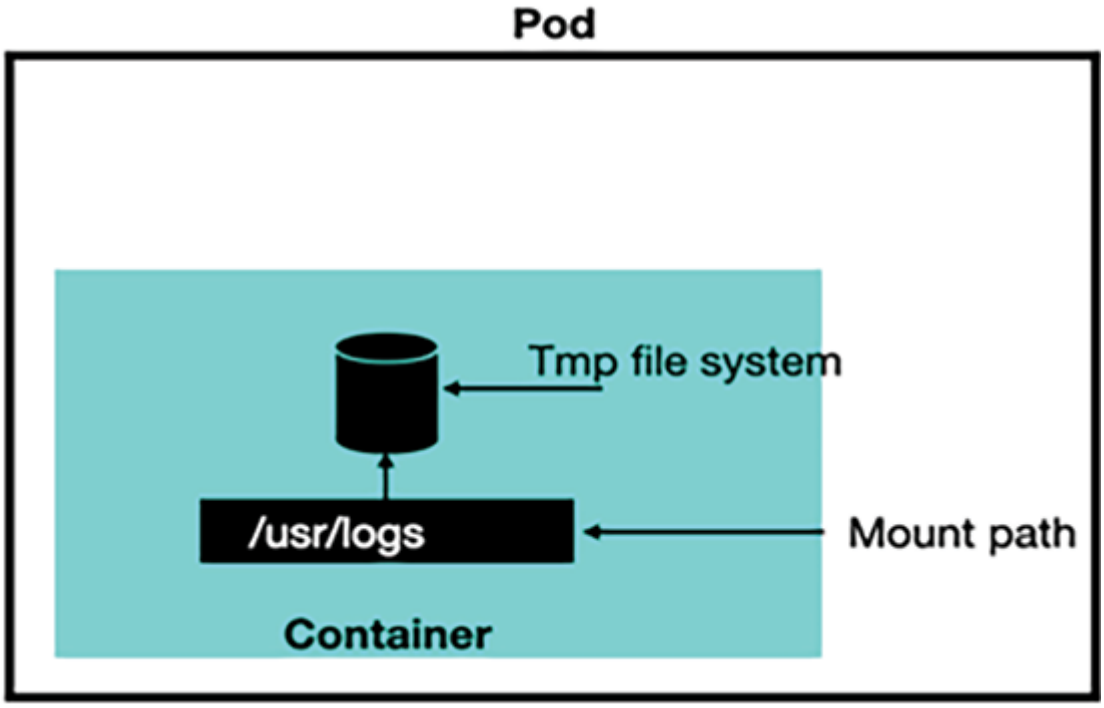


Figure 7.1: Temporary file system in Container

As we have seen in the previous chapter on multi-container Pods, a Pod can consist of one or more containers. Volumes are basic storage units of Kubernetes, and in Kubernetes, these volumes are part of Pod.

Refer to the following Figure which shows Pod volume:

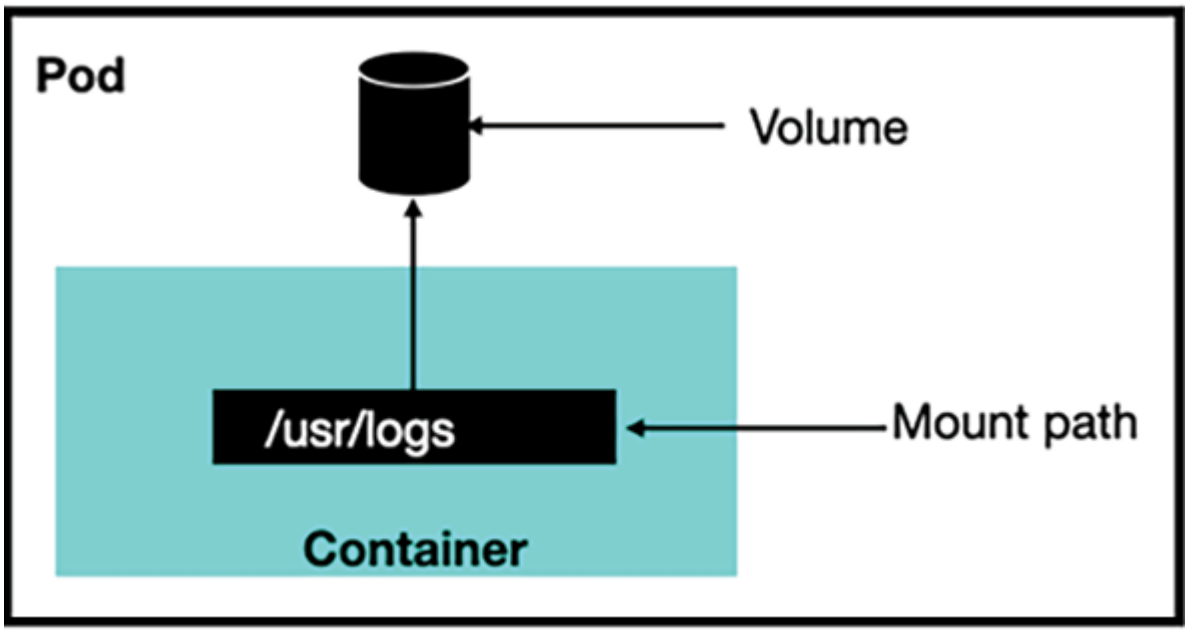


Figure 7.2: Pod volumes

There are two types of volumes in Kubernetes, ephemeral volume, which exist till the Pod is alive, and persistence volume, which outlives the Pod.

The following table below mentions the different types of volumes and its description:

Table 7.1: Volume types

Pods and deployments volume configuration

To configure the volumes to Pods below steps need to be performed:

Declare a volume type using attribute spec.volume

You need to mount the volume to the path of the consuming container using attribute

The following example shows how a volume type of an empty directory is first defined (the name of this volume is and then how the container mounts the volume in mount path

New files and directories can be created without constraint in an empty directory.

```
apiVersion: v1

kind: Pod

metadata:

  name: volume-app

spec:
```

```
  volumes:

- name: logvol

   emptyDir: {}


  containers:

- image: nginx

   name: nginx

   volumeMounts:

   - mountPath: /var/logs

      name: logvol
```

Once you create the Pod using the manifest file, you can query the container and navigate to The Volume type used in this example is emptyDir which initializes the mountpath as an empty directory where you can create files.

Refer to the following Figure which shows the Pod created using the empty Dir

```
kishore@ss-MacBook-Pro chapter-7 % kubectl create -f logvol.yaml
pod/volume-app created
kishore@ss-MacBook-Pro chapter-7 % kubectl get po
NAME         READY   STATUS    RESTARTS   AGE
volume-app   1/1     Running   0          17s
kishore@ss-MacBook-Pro chapter-7 % kubectl exec volume-app -it -- /bin/sh
# cd /var/logs
# touch one.txt
# ls
one.txt
```

Figure 7.3: Pod created using empty dir volume

Persistent Volume and PersistentVolumeClaim

You have seen in the previous section how to create volume locally in Pod specification. In a real production environment where you need flexibility and decoupling, you will not create local volume but a Persistent volume. A Persistent Volume is an independent resource that connects to external storage. An independent resource means it runs separately from any Pod in your Kubernetes environment. PV can point to all persistent storage types which outlive the Pod. PV does not belong to any Kubernetes namespace; it is a cluster-level resource similar to the node.

Persistent volume work in tandem with PersistentVolumeClaim A persistent volume is a storage resource in the Kubernetes cluster, while a PVC is a request for storage.

Typically, in production, Kubernetes cluster PVs are created by the administrator, and the development team creates PVC. So, in PV definition, you define the capacity of the volume and its access right; in PVC, you define the minimum capacity size and access rights of PV needed by the Pod.

Let us understand this with an example. The following is the manifest file of the definition of PV:

apiVersion: v1

kind: PersistentVolume

```
metadata:

name: pvdemo

spec:

capacity:

  storage: 2Gi

accessModes:

  - ReadWriteOnce

hostPath:

  path: “/mnt/data”
```

Capacity defines the storage size; access mode defines the access to the volume. For example, ReadWriteonce allows only a single node to access the volume in read-write mode. If you want to create this resource using the manifest file, you can use the kubectl create command to create the persistent volume in the Kubernetes resource.

Let us look at the PersistentVolumeClaim manifest:

```
apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: pvcdemo

spec:

accessModes:


  - ReadWriteOnce

volumeMode: Filesystem

resources:

  requests:

    storage: 1Gi
```

In PVC, you define the access modes and capacity requests. So, at run time, it is the job of the Kubernetes cluster (in actual Kubernetes master plane) to find a PV that matches the PVC and binds the two. Similar to PV, use create kubectl manifest file command to create PVC.

Configure Pod with PV and PVC

In the previous section, we saw what PV and PVC are. In this section, let us see how this PVC is configured in Pod.

Let us define PV and PVC as manifest YAML.

The following example is the YAML manifest file of PV This manifest file will create a persistent volume with accessmodes ReadWriteOnce, and storage capacity of 2Gi, and host path

```
apiVersion: v1

kind: PersistentVolume

metadata:

name: pvdemo

spec:

capacity:

  storage: 2Gi

accessModes:
```

- ReadWriteOnce

hostPath:

path: “/usr/local”

Create the persistent volume using the following command:

$ kubectl create -f pv.yaml

The above command creates a persistent volume.

Refer to the following Figure which shows the persistent volume created:

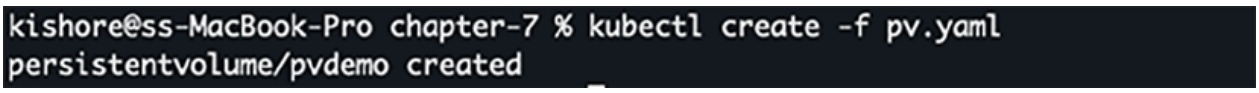


Figure 7.4: Creating persistent volume

Execute the below command to view the persistent volume:

$ kubectl get pv

The above command shows the persistent volume in the cluster.

The following Figure 7.5 shows the persistent volume in the cluster:

```
kishore@ss-MacBook-Pro chapter-7 % kubectl get pv
NAME     CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM   STORAGECLASS   REASON   AGE
pvdemo   2Gi        RWO            Retain           Available                                   3m15s
```

Figure 7.5: Persistent volume in the cluster

As shown in Figure the PV is available and is not mounted yet.

The following example is the YAML manifest file of PVC

This manifest creates a PersistentVolumeClaim that requests the Persistent volume pvdemo, which we created in the previous step:

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: pvcdemo

spec:

accessModes:

- ReadWriteOnce

volumeMode: Filesystem

```
resources:
  requests:
    storage: 1Gi
```

The following command creates a

```
$ kubectl create -f pvc.yaml
```

Refer to the following Figure which shows the PersistentVolumeClaim created:

```
kishore@ss-MacBook-Pro chapter-7 % kubectl create -f pvc.yaml
persistentvolumeclaim/pvcdemo created
```

Figure 7.6: PersistentVolumeClaim created

The following command shows the PersistentVolumeClaim in the cluster:

```
$ kubectl get pvc
```

The following Figure 7.7 shows the PersistentVolumeClaim in the cluster. As you can see, the status of the PVC is bound to the volume:

```
kishore@ss-MacBook-Pro chapter-7 % kubectl get pvc
NAME      STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
pvcdemo   Bound    pvc-7c68fbad-ed7d-4925-aa3c-b9d7fdc65f7e   1Gi        RWO            standard       28s
```

Figure 7.7: PersistentVolumeClaim in the cluster

The next step is to create a new Pod and associate the PVC created in the previous step. Let us create a new Pod named app with an image http-server that will be used to mount the PersistentVolumeClaim with the path

```
apiVersion: v1

kind: Pod

metadata:

  name: app

spec:

  volumes:

    - name: demovol

      persistentVolumeClaim:

        claimName: pvcdemo

  containers:

    - name: democontainer

      image: nginx
```

```
    ports:
      - containerPort: 80
        name: “http-server”
    volumeMounts:
      - mountPath: “/usr/share/nginx/html”
        name: demovol
```

With the above manifest, YAML creates a Pod using kubectl create Pod command.

The following Figure 7.8 shows Pod attached to PVC running in the cluster:

```
kishore@ss-MacBook-Pro chapter-7 % kubectl get po
NAME   READY   STATUS    RESTARTS   AGE
app    1/1     Running   0          8m39s
```

Figure 7.8: Pod created with PVC

$Kubectl describe pvc pvcdemo

The above command describes PersistentVolumeClaim, which now shows the Pod that mounted it.

As shown in the following Figure the Pod app is using the PVC

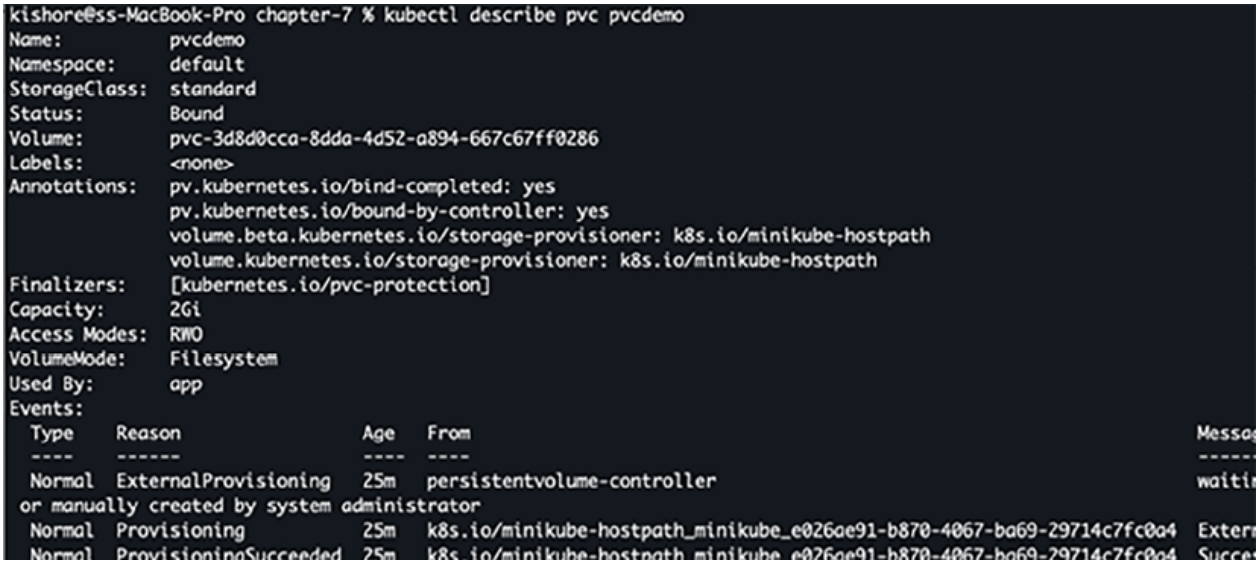


Figure 7.9: Describe PVC shows the mounted Pod

Storage class

You use a storage class when you want to dynamically provision PV in a Kubernetes cluster.

Storage classes help Pod to dynamically ask for specific types of storage as per demand. Provisioner, parameters, and reclaim policy are storage class fields used when a persistent volume belonging to the class needs to be dynamically provisioned.

Sample manifest file of the storage class is described below:

```
apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: gp2std

provisioner: kubernetes.io/aws-ebs

parameters:

type: gp2
```

reclaimPolicy: Retain

volumeBindingMode: Immediate

The following command creates the storage class from the manifest file:

$Kubectl create -f storage.yaml

Refer to the following Figure 7.10 to create a storage class:

```
kishore@ss-MacBook-Pro chapter-7 % kubectl create -f store.yaml
storageclass.storage.k8s.io/gp2std created
```

Figure 7.10: Create storage class

$Kubectl get Storageclass

The above command shows the storage class in the cluster.

Refer to the following Figure which shows the storage class in the cluster:

```
kishore@ss-MacBook-Pro chapter-7 % kubectl get storageclass
NAME                PROVISIONER             RECLAIMPOLICY   VOLUMEBINDINGMODE   ALLOWVOLUMEEXPANSION   AGE
gp2std              kubernetes.io/aws-ebs   Retain          Immediate           false                  52s
```

Figure 7.11: Get storage class from cluster

Conclusion

In this chapter, we looked at the various Kubernetes volumes, different volume types, and how to configure Persistent volume and Persistence volume claims with Pod.

The next chapter will look into configuring volume environment variables with ConfigMap and secret.

Questions

Create a Pod YAML manifest file that has two containers with image nginx. Define a Volume of type emptyDir for the Pod. Mount the Volume to containers 1 and 2 to path

Create a PersistentVolume named which maps to the hostPath Set the access mode to Storage capacity is 5Gi. Ensure that the status of the PersistentVolume shows

Create a PersistentVolumeClaim named The access to this PVC is Request a capacity of 2Gi. Once the PVC is created, ensure that the status of the PersistentVolume shows

Also, mount the PersistentVolumeClaim in a Pod running the image nginx at the mount path

Solution

Solution for question 1:

Use the below defined YAML manifest file to create Pod with two containers, container 1 and container 2

```
apiVersion: v1

kind: Pod

metadata:

  name: ng

spec:

  containers:

  - image: nginx

    name: container1

  - image: nginx
```

name: container2

Edit the YAML file to add in the volumes and mount path to: /usr/share/nginx/html

```
apiVersion: v1

kind: Pod

metadata:



  name: ng

spec:

volumes:

  - name: shared-vol

    emptyDir: {}

  containers:

  - image: nginx

    name: container1
```

```
    volumeMounts:
    - name: shared-vol
      mountPath: /usr/share/nginx/html
  - image: nginx
    name: container2
    volumeMounts:
    - name: shared-vol
      mountPath: /usr/share/nginx/html
```

Use the kubectl create command to create the Pod with two containers, container 1 and container 2, which have mount paths to /usr/share/nginx/html

Solution for question 2:

Use the below defined YAML manifest file to create PV

```
kind: PersistentVolume
apiVersion: v1
```

```
metadata:
  name: demopv
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /usr/local
```

Execute the following command to display pv in the cluster, which shows the status of pv as available:

```
$ kubectl get pv
```

Solution for question 3:

Use the below defined YAML manifest file to create PVC demopvc

```
kind: PersistentVolumeClaim

apiVersion: v1

metadata:

  name: demopvc

spec:

  accessModes:

    - ReadWriteOnce

  resources:

    Requests:

      storage: 2Gi
```

Execute the following command to display pvc in the cluster, which shows the status of pvc as bound:

```
$ kubectl get pvc
```

Solution for question 4:

Attach the PVC to the Pod using the below manifest file. Once you create the manifest file, use the kubectl create Pod command to create the Pod:

```
apiVersion: v1

kind: Pod



metadata:

  name: nginx

spec:

  volumes:

    - name: logs-volume

      persistentVolumeClaim:

        claimName: demopvc

  containers:

  - image: nginx

    name: nginx
```

```
volumeMounts:

  - mountPath: “/var/log/nginx”

    name: logs-volume
```

Once Pod is created, use the following command to describe PersistentVolumeClaim, which now shows the Pod ngix that mounted it:

```
$Kubectl describe pvc demopvc
```

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 8

Configuring Volume, Environment Variables, ConfigMap, and Secrets

Introduction

This chapter will teach us some advanced concepts to configure the Kubernetes object. We will first start with how to configure runtime behavior using environment variables. How to configure apps through ConfigMaps and how to handle sensitive information in Kubernetes using secrets. We will also cover how to mount to Pods ConfigMaps and secrets as environment variables and volumes. So, let us get started.

Structure

In this chapter, we will cover the following topics:

Kubernetes configuration

Environment variables

Define and consume configuration data

ConfigMaps

Secrets

Mounting volume and environment variables to Pods

Summary

Objectives

The objective of this chapter is to ensure you have a good understanding of configuring Kubernetes objects' runtime behavior using environment variables. Learn how to use ConfigMaps and handle sensitive information in Kubernetes using secrets.

Kubernetes configuration

Configuration in the context of this chapter means how to configure Pods. This chapter will teach us some advanced concepts to configure the Pods.

Most of the applications have some or other configuration data. For example, it could be the username and password of a database connection, API endpoint to an external web service, and so on. It is not good practice to tightly bundle these configuration data to the code as you need to rebuild the image for any change in the configuration information. Also, it is a bad practice to build separate target Docker images for each target environment, like one for development and a separate one for production. Application image should be built once, and the configuration information, which will be different for each environment, be it development or production, should be externalized and injected at runtime while instantiating the image.

Kubernetes provides different ways to configure their runtime behavior via environment variables, ConfigMaps, and secrets. We will be looking at that in detail in the next few sections.

Environment variables

Kubernetes allows you to define the environment variables for each container at Pod.

Adding environment variables to the Kubernetes definition is quite straightforward. Let us look at it with an example.

The manifest YAML file below shows how to declaratively add environment variables to the container. The following is the manifest YAML named

```
apiVersion: v1

kind: Pod

metadata:

  name: nginx-env

spec:

  containers:

  - image: nginx

    name: cont1
```

```
env:

- name: PROFILE

  value: prod

- name: VERSION


  value: '1.0'
```

The YAML has two environment variables defined: which has the value and which has the value 1.0. Hopefully, by now, you are comfortable creating Kubernetes objects declaratively.

Refer to Figure which shows how the Kubernetes object with the environment variable is created:

```
kishore@ss-MacBook-Pro chapter8 % kubectl create -f env.yaml
pod/nginx-env created
kishore@ss-MacBook-Pro chapter8 % kubectl get po
NAME        READY   STATUS    RESTARTS   AGE
nginx-env   1/1     Running   0          18s
```

Figure 8.1: Pod with environment variables

Once these environment variables are defined at the container level, the application can use these environment variables referring to the variable names. You can query the Pod and list the environment variables, and you will see that the recently created variable is part of it.

$kubectl exec nginx-env -- printenv

The above command will print the environment variable of the Pod on the screen.

Refer to Figure which lists the environment variable PROFILE and

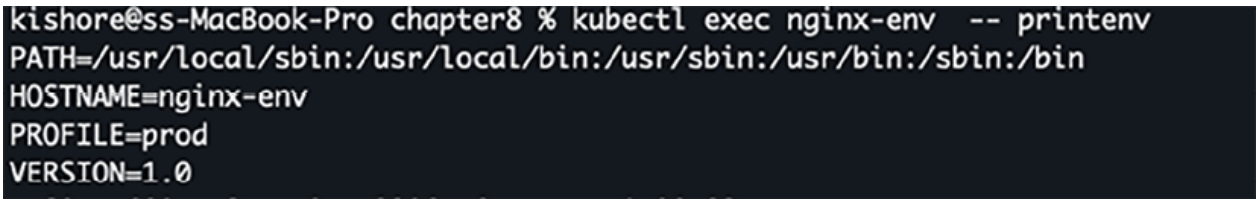


Figure 8.2: List down the environment variables

One issue with environment variables is that these are hardcoded within Pods. Thus, you need separate pod definitions for different environments, for example, one for the production Pod and a separate one for the development Pod.

If you want to reuse the same Pod definition across different environments, you need to decouple the configuration from the pod descriptor. ConfigMap can be used in this scenario which will be used as a source for the environment variable. We will look at more details in the subsequent section.

Define and consume configuration data

In today's world of continuous delivery, the artifact (Docker image) for each of the single source code commits must be only made once. The Docker image should be stored in a repository like JFrog or Nexus. The same image should be downloaded in different run time environments like development, testing, and production. Configuration data needed in these runtime environments will differ and need to be injected into runtime environments during image instantiation. In Kubernetes, there are two ways to define configuration data: ConfigMaps and secrets. Both options are used to change the configuration data of the Pods without the need to redeploy the Pod. ConfigMap and secrets are key-value pairs that can be injected into the container as volume or environment variables. We will look at the secret and ConfigMap in the next two sections.

ConfigMaps

One simplest way to create ConfigMap is using the kubectl create config map command.

$ kubectl create configmap test-config --from-literal=interval=10

The above command creates a configmap named test-config with a single-entry

Refer to Figure which creates ConfigMap named

```
kishore@ss-MacBook-Pro chapter8 % kubectl create configmap test-config --from-literal=interval=10
configmap/test-config created
kishore@ss-MacBook-Pro chapter8 % kubectl get configmap test-config
NAME          DATA   AGE
test-config   1      16s
```

Figure 8.3: ConfigMap created with single entries

To create ConfigMap with multiple entries

$ kubectl create configmap config2 --from-literal=interval=10 --from-literal=status=true

The above command creates a configmap named config2 with multiple entries interval=10 and

Refer to Figure which creates configMap named config2 with multiple entries:

```
kishore@ss-MacBook-Pro chapter8 % kubectl create configmap config2 --from-literal=interval=10 --from-literal=status=true
configmap/config2 created
kishore@ss-MacBook-Pro chapter8 % kubectl get configmap config2
NAME      DATA   AGE
config2   2      9s
```

Figure 8.4: ConfigMap created with multiple entries

You can also create ConfigMap declaratively. The YAML description is shown below, YAML file is saved as

```
apiVersion: v1

kind: ConfigMap

metadata:

  name: userconfig

data:

    user: Kishore
```

Refer to Figure which creates configMap named userconfig declaratively:

```
kishore@ss-MacBook-Pro chapter8 % kubectl create -f userconfig.yaml
configmap/userconfig created
kishore@ss-MacBook-Pro chapter8 % kubectl get configmap userconfig
NAME         DATA   AGE
userconfig   1      19s
```

Figure 8.5: ConfigMap created declaratively from YAML file

There are other ways to create ConfigMap. You can create configMap also from the contents of a file.

For this, first create a file, in the following example, create a JSON file The values file can be any format, for example, text format. In this example, we have considered json file:

{

user: dbsuser

db_url: jdbc://mysql://localhost/test

}

To create ConfigMap from a file

$kubectl create configmap dbconfig --from-file=values.json

The above command creates a configMap with values from file

Refer to Figure which creates ConfigMap from the file:

```
kishore@ss-MacBook-Pro chapter8 % kubectl create configmap dbconfig --from-file=values.json
configmap/dbconfig created
kishore@ss-MacBook-Pro chapter8 % kubectl get configmap dbconfig
NAME       DATA   AGE
dbconfig   1      10s
```

Figure 8.6: ConfigMap created from file

Refer to Figure which describes the ConfigMap created from file:

```
kishore@ss-MacBook-Pro chapter8 % kubectl describe configmap dbconfig
Name:         dbconfig
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
values.json:
----
{

user: dbsuser
db_url: jdbc://mysql://localhost/test

}


BinaryData
====
```

Figure 8.7: Details of ConfigMap created from file

You can also create a ConfigMap with contents in a file and treat it as env variables.

For this, first, create a file as shown below, which has a key-value pair. As shown in the example below creates values.env, which has key-value pair.

db_url= jdbc://mysql://localhost/test

Execute the following kubectl command to create configmap with contents in a file and treat it as env variables.

$kubectl create configmap envdbconfig --from-env-file=values.env

The above command creates

You can view the configmap YAML by executing the following command:

$ kubectl get configmap envdbconfig -o yaml

The above command outputs the YAML definition of ConfigMap created.

Refer to Figure which describes the ConfigMap created from the env file and its output of it in YAML format:

```
kishore@ss-MacBook-Pro chapter8 % kubectl create configmap envdbconfig --from-env-file=values.env
configmap/envdbconfig created
kishore@ss-MacBook-Pro chapter8 % kubectl get configmap envdbconfig -o yaml
apiVersion: v1
data:
  dburl: jdbc://mysql://localhost/test
kind: ConfigMap
metadata:
  creationTimestamp: "2023-05-07T08:43:12Z"
  name: envdbconfig
  namespace: default
  resourceVersion: "106044"
  uid: ed7757b2-4c7e-4612-97b3-3d10937dd57a
```

Figure 8.8: ConfigMap created from env file

In the previous examples, we have seen creating ConfigMap from a file, so instead of importing a single file at a time, you can also import all the files in a directory.

For this case, first, create a directory and place your txt files with name values.

For this example, the directory is created under configdir and it has two .txt files:

First txt file db.txt with value:

user=abc

Second file server.txt with values:

connection_pool=10

connection_type=jdbc

Execute the following kubectl command to create configmap with contents from files in the directory

$kubectl create configmap dirconfig --from-file=configdir

The above command creates ConfigMap.

Refer to Figure which describes the ConfigMap created from the directory containing files:

```
kishore@ss-MacBook-Pro chapter8 % kubectl create configmap dirconfig --from-file=configdir
configmap/dirconfig created
kishore@ss-MacBook-Pro chapter8 % kubectl get configmap dirconfig -o yaml
apiVersion: v1
data:
  db.txt: |+
    user=abc

  server.txt: |
    connection_pool=10
    connection_type=jdbc
kind: ConfigMap
metadata:
  creationTimestamp: "2023-05-07T08:57:54Z"
  name: dirconfig
  namespace: default
  resourceVersion: "106749"
  uid: a0db12f5-befc-4920-add2-f17bca4f4f86
```

Figure 8.9: ConfigMap created from contents of files in directory

Secrets

All the examples that we have seen for now are on how to store non-sensitive non-secure information, and ConfigMap does it well. You need to use secrets if the information that needs to be stored is secure and sensitive. A good example is any password used in the application, you do not want to be exposed as a clear text for others to see.

Like ConfigMap secrets hold name-value The way Kubernetes makes the secrets secure is the secrets are only passed to the nodes that run Pods that use secrets.. Also secrets are stored within the nodes memory and are never written to disk.

Similar to configMap, secrets can be created imperatively using kubectl create secret command.

$kubectl create secret generic dbcredentials --from-literal=dbpwd=test@123

The above command creates secret dbcredentials, which the database password as test@123, and outputs it in YAML format.

Refer to Figure which creates secret and outputs it in YAML format. You will see the values of dbpwd are in base64 format and not in clear text:

Figure 8.10: Secret created from literal

Similar to ConfigMap, you can create secret from file. You can also create secret declaratively, but the value needs to be base64 encoded for type of opaque as shown in below example:

apiVersion: v1

kind: Secret

metadata:

name: databsecredentials

type: Opaque

data:

dbpwd: dGVzdEAxMjM=

Mounting volume and environment variables to Pods

In previous sections, we have seen how to create secret and ConfigMap,. In this section, we will see how to mount the configMap and secret to the Pod. There are two ways to do it:

Consuming configMap and secrets as environment variables.

Mounting configMap and Secrets as volume.

Let us first see how configMap and secret are consumed as environment variables.

First, create configMap and secret with the below kubectl command:

kubectl create configmap db-username --from-literal=username=admin

kubectl create secret generic db-password --from-literal=pwd=test@123

The above commands create ConfigMap and secrets from literals.

Refer to Figure which creates ConfigMap and secret:

```
kishore@ss-MacBook-Pro chapter8 % kubectl create secret generic db-password --from-literal=pwd=test@123
secret/db-password created
kishore@ss-MacBook-Pro chapter8 % kubectl create configmap db-username --from-literal=username=admin
configmap/db-username created
```

Figure 8.11: Create configmap and secrets from literals

Once ConfigMap and secret are created in the declarative Pod definition using envFrom.configMapRef and envFrom.secretRef as shown in the following YAML file

```
apiVersion: v1

kind: Pod

metadata:

  name: testpod

spec:

  containers:

  - image: nginx

    name: app

    envFrom:

    - secretRef:

        name: db-password
```

- configMapRef:

    name: db-username

Use kubectl create command to create the Pod. Refer to Figure 8.12 which creates Pod referencing to the ConfigMap and secret:

```
kishore@ss-MacBook-Pro chapter8 % kubectl create -f config-secret-env.yaml
pod/testpod created
kishore@ss-MacBook-Pro chapter8 % kubectl get po testpod
NAME      READY   STATUS    RESTARTS   AGE
testpod   1/1     Running   0          9s
```

Figure 8.12: Create Pod mounting ConfigMap and secret

Use kubectl exec command to query the Pod, which prints the environment variables, and you will see the environment variables defined via ConfigMap and secret to be present.

Refer to the following Figure which queries the Pod for environment values:

```
kishore@ss-MacBook-Pro chapter8 % kubectl exec testpod -- env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=testpod
pwd=test@123
username=admin
KUBERNETES_SERVICE_HOST=10.96.0.1
KUBERNETES_SERVICE_PORT_HTTPS=443
```

Figure 8.13: Query Pod for environment variables

Also, secrets and configMap can be mounted as a volume. Once mounted as volume the application can read from the key-value pair from the file system

from the expected mount path.

We will use the same secret and ConfigMap which was used for earlier environment variable examples. Below is the YAML manifest file of mounting ConfigMap and secret as volume. The YAML can be used to create Pod Config pod using kubectl command:

```
apiVersion: v1

kind: Pod

metadata:

  name: configpod

spec:

  containers:



  - image: nginx

    name: appvol

    volumeMounts:

    - name: config-volume

      mountPath: /etc/config
```

```
  - name: secret-volume
    mountPath: /etc/config1
volumes:
- name: config-volume
  configMap:
    name: db-username
- name: secret-volume
  secret:
    secretName: db-password
```

Secrets mounted as the volume will expose their values in Base64-decoded form. You can easily verify the value of secret and ConfigMap by opening an interactive shell.

Refer to Figure which queries the volume pod for ConfigMap values:

```
kishore@ss-MacBook-Pro chapter8 % kubectl exec -it configpod -- /bin/sh
# cat /etc/config/username
admin#
```

Figure 8.14: Query Pod for ConfigMap values

Refer to Figure which queries the volume pod for secret values:

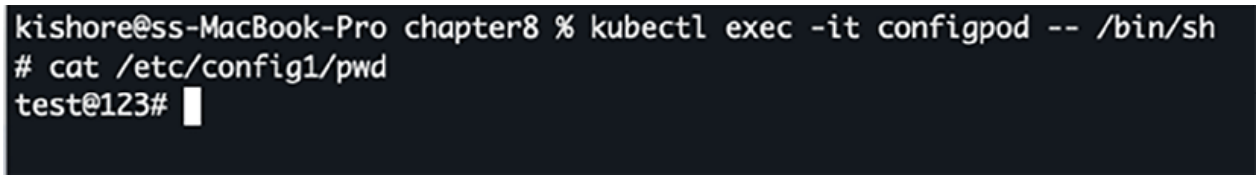


Figure 8.15: Query Pod for secret values

Conclusion

In this chapter, we looked at some advanced concepts to configure Kubernetes object like ConfigMap and secrets, and how to mount these Pods using environment variables and volume.

In the next chapter, we will look into how to define security context, privilege, and access control and how to assign a service account to a Pod.

Questions

Use the imperative approach to create a configMap named Feed in the key-value pairs username=bot as literals.

Use the imperative approach to create a Secret named Feed in the key-value pairs pwd=test123 literals.

Create a Pod named consumer with the image nginx and mount the secret as a Volume with the mount path Open an interactive shell and inspect the values of the secret.

Inject the ConfigMap values of democonfigmap into the existing Pod as environment variables.

Solutions

Solution for question 1:

The below kubectl create command is a declarative way to create configmap:

$ kubectl create configmap democonfigmap --from-literal=username=bot

Once configMap is created you can check its status with the following command:

$kubectl get configmap democonfigmap

Solution for question 2:

The below kubectl create command is a declarative way to create secret:

$kubectl create secret generic demosecret --from-literal=pwd=test123

Once secret is created you can check its status with below command:

$kubectl get secret demosecret

Solution for question 3:

Below is the Pod manifest with secret mounted as volume:

```
apiVersion: v1


kind: Pod

metadata:

  name: consumer

spec:

  containers:

  - image: nginx

    name: appvol

    volumeMounts:

    - name: secret-volume

      mountPath: /var/app

volumes:
```

```
  - name: secret-volume

    secret:

      secretName: demosecret
```

You can create the pod using regular kubectl create -f < yaml file name>

Once Pod is created, you can query the Pod using the following command:

$kubectl exec -it consumer -- bin/sh

```
# cat /var/app/pwd
```

The above command will display the value of secret

Solution for question 4:

You need to re-create the consumer Pod to add the configMap as environment variables, as Kubernetes does not allow adding new environment variables to a running container.

The first step is to delete the existing consumer pod:

$kubectl delete po consumer

Then create the below YAML to add the environment variable definition. The following is an update to the YAML definition:

```
apiVersion: v1

kind: Pod

metadata:

  name: consumer

spec:

  containers:

  - image: nginx

    name: appvol

    volumeMounts:


    - name: secret-volume

      mountPath: /var/app

    envFrom:
```

```
    - configMapRef:
        name: democonfigmap
  volumes:
  - name: secret-volume
    secret:
      secretName: demosecret
```

You can create the pod using regular kubectl create -f < yaml file

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 9

Service Accounts, Resource Quota and Security Contexts

Introduction

This chapter will teach us how to define security context, privilege, access control settings, and the configuration needed to assign a service account to a Pod. We will also learn about resource boundaries and how to set resource quotas.

Structure

In this chapter, we will cover the following topics:

Define security context

Resource boundary

Resource quota

Service account

Objectives

The objective of this chapter is to ensure you have a good understanding of defining security contexts, privilege, and access control settings and to learn about resource boundaries to set resource quotas and service accounts. So, let us get started.

Define security context

Security context defines privilege and access control settings for a Pod or container. By default, the container runs with root privileges. You can use kubectl explain, which shows the different security contexts that can be set, like, file system group.

Refer to Figure which shows the different contexts:

```
kishore@ss-MacBook-Pro chapter9 % kubectl explain pod.spec.securityContext
KIND:     Pod
VERSION:  v1

RESOURCE: securityContext <Object>

DESCRIPTION:
     SecurityContext holds pod-level security attributes and common container
     settings. Optional: Defaults to empty. See type description for default
     values of each field.

     PodSecurityContext holds pod-level security attributes and common container
     settings. Some fields are also present in container.securityContext. Field
     values of container.securityContext take precedence over field values of
     PodSecurityContext.

FIELDS:
   fsGroup      <integer>
     A special supplemental group that applies to all containers in a pod. Some
     volume types allow the Kubelet to change the ownership of that volume to be
     owned by the pod:

     1. The owning GID will be the FSGroup 2. The setgid bit is set (new files
     created in the volume will be owned by FSGroup) 3. The permission bits are
```

Figure 9.1: Security context details

Security contexts need to be applied declaratively; there is no easy way to apply security context settings imperatively while working with Kubernetes.

Let us try to understand the security context with an example.

The YAML file below has security context defined at the Pod level. Let us try to understand the various security context options in this example.

It is the Group ID to run the entry point of the container process.

It is the User ID to run as the entry point of the container process.

It is the secondary or supplement group.

Also, in the following example, the YAML file. securityContext is set to allowPrivilegeEscalation: which means we do not do any privilege escalation at the container level:

```
apiVersion: v1

kind: Pod

metadata:

  name: securitydemo

spec:
```

```
securityContext:
  runAsUser: 1000
  runAsGroup: 3000
  fsGroup: 2000
volumes:
- name: secvol


  emptyDir: {}
containers:
- name: cont1
  image: busybox
  command: [ “sh”, “-c”, “sleep 1h” ]
  volumeMounts:
  - name: secvol
    mountPath: /data/demo
```

```
    securityContext:

      allowPrivilegeEscalation: false
```

With the above YAML manifest file, let us create the Pod declarative and try to see how the security context comes into play.

So, the first command is to create the Pod using manifest YAML:

```
$kubectl create -f secdemo.yaml
```

Once created, let us query the Pod and jump inside the Pod using the following command:

```
$kubectl exec -it securitydemo –-sh
```

Once you log into the Pod and execute command which lists the active process, we get to see processes running as UID as 1000 (defined earlier in YAML security context definition).

Next, we go to the data directory and execute the following command:

```
$ ls -l
```

The above command lists the files and directory and shows the data directory. The group owner of this directory is set to group 2000. That is a security context as well. And if you go into the demo sub-directory and we use the following command to create the file and list the file

$ touch test.txt

$ ls -l

Then we can see the ownership of the file test.txt set to 1000, 2000, exactly what we specified in the security contexts.

Refer to the following Figure which lists all the steps defined above:

```
kishore@ss-MacBook-Pro chapter9 % kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
securitydemo   1/1     Running   0          8m56s
kishore@ss-MacBook-Pro chapter9 % kubectl exec -it securitydemo -- sh
~ $
~ $ ps
PID   USER     TIME  COMMAND
    1 1000      0:00 sh -c sleep 1h
   36 1000      0:00 sh
   42 1000      0:00 ps
~ $ cd /data
/data $ ls -l
total 4
drwxrwsrwx    2 root     2000          4096 May 13 16:34 demo
/data $ cd demo
/data/demo $ touch test.txt
/data/demo $ ls -l
total 0
-rw-r--r--    1 1000     2000             0 May 13 16:44 test.txt
```

Figure 9.2: Security context implementation of Pod

Also, note that security settings defined on the Pod level are applicable for all the containers running in the Pod. For example, the runAsUser feature is applied to all containers running in the Pod. You can always override the Pod-level settings by defining context at the container level; container-level settings will take precedence and can override the Pod-level settings.

For the following Pod definition of nginx declarative YAML, the runAsNonRoot is set to true.

As per Kubernetes spec definition, runAsNonRoot Indicates that the container must run as a non-root user. If true, the Kubelet will validate the image at runtime to ensure that it does not run as UID 0 (root) and fail to start the container if it does.

Let us look at one more example; in this example, let us look at the feature

```
apiVersion: v1

kind: Pod

metadata:

  name: rootcheck

spec:

  containers:

  - image: nginx:1.18.0

    name: seccontainer



    securityContext:
```

runAsNonRoot: true

Let us create this Pod declaratively. You will notice the container is not created, and the container fails during the startup process with the status This is because the container needs the root privilege to run itself.

Refer to the following Figure which shows the container creation error:

kishore@ss-MacBook-Pro chapter9 % kubectl create -f rootcheck.yaml
pod/rootcheck created
kishore@ss-MacBook-Pro chapter9 % kubectl get po
NAME READY STATUS RESTARTS AGE
rootcheck 0/1 CreateContainerConfigError 0 3s
kishore@ss-MacBook-Pro chapter9 % kubectl describe po rootcheck | grep -i runAsNonRoot
Warning Failed 1s (x3 over 14s) kubelet Error: container has runAsNonRoot and image will run as root
_default(f4544e09-1709-4091-9711-1da619e977bc)", container: seccontainer)

Figure 9.3: Container creation error

To resolve this, you need to set the runAsNonRoot to false or look for the nginx image, which does not need root privilege to run.

Resource boundary

Kubernetes objects like namespaces do not enforce how much computing resources like CPU, memory, or disk space each Pod or container should consume. They also do not limit the number of Kubernetes objects that can be created. So theoretically, Kubernetes objects can consume unlimited resources till the maximum available capacity is reached. The next topic, resource quota, is important to understand how we can limit the computing resources consumed by Pod or container.

Resource quota

In this topic, we learn how to use resource limitations and quota in Kubernetes to determine the behavior and resource allocation. By default, a Pod will use as much CPU and memory as necessary to do its work, and this is because Pod is starting a container and a process that runs on Linux. By default, Linux does not do anything to limit the number of resources available to processes.

Kubernetes resource quota helps to establish the maximum usable number of resources per namespace. Resource quota helps to define the following as part of the namespace:

Setting an upper limit for the number of Kubernetes objects that can be created for a specific type (For example, number of Pods, ConfigMaps, or secrets).

Limiting the total sum of computing resources in terms of CPU and memory.

In any organization, resource quotas are typically managed by a Kubernetes administrator.

To create a Kubernetes resource quota, execute the following command. As discussed, the resource quota is applied at the namespace. The first step is to create a namespace and then apply resource quota to the namespace:

$ kubectl create ns resource-test

The above command creates a namespace resource test.

Refer to Figure which shows the command to create a namespace resource test. We had already seen how to create a namespace in Chapter Core Concepts - Pods, Deployments, ReplicaSet, and

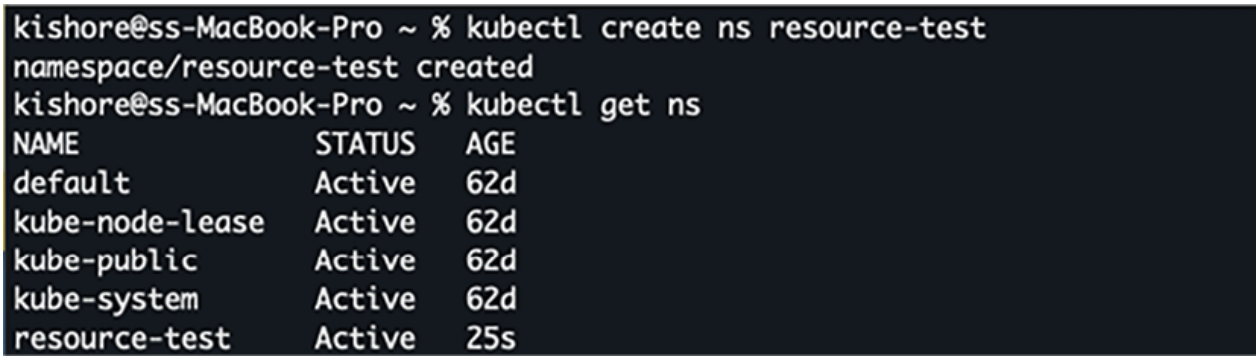


Figure 9.4: Create namespace resource test.

The next step is to create a resource quota in YAML. Let us create a resource quota with the following constraints:

Limit the number of Pods in the namespace resource test to 1.

Define the minimum CPU resources requested by a Pod in the namespace resource test to 1 CPU.

Define the minimum memory resources requested by a Pod in the namespace resource test to 1 GB.

Define the maximum resources a Pod uses in the namespace resource test to 2 CPUs and 2 GB of RAM.

The following is the mentioned YAML for Let us create the resource quota in the namespace resource-test declaratively:

```
apiVersion: v1

kind: ResourceQuota

metadata:

  name: res-quota

spec:

  hard:

    pods: 1

    requests.cpu: “1”

    requests.memory: 1024m

    limits.cpu: “2”

    limits.memory: 2096m
```

The following command will create a resource quota in the namespace

$kubectl create -f resource.yaml --namespace=resource-test

The following command will describe the newly created resource quota:

$kubectl describe resourcequota res-quota --namespace=resource-test

Refer to the following Figure which shows the newly created resource quota:

```
kishore@ss-MacBook-Pro chapter9 % kubectl create -f resource.yaml --namespace=resource-test
resourcequota/res-quota created
kishore@ss-MacBook-Pro chapter9 % kubectl describe resourcequota res-quota --namespace=resource-test
Name:           res-quota
Namespace:      resource-test
Resource        Used  Hard
--------        ----  ----
limits.cpu      0     2
limits.memory   0     2096m
pods            0     1
requests.cpu    0     1
requests.memory 0     1024m
```

Figure 9.5: Create and describe resource quota

The next step is to validate the resource quota created. This is to validate if the constraints defined as part of resourcequota are enforced.

Let us create a Pod with the following manifest YAML:

apiVersion: v1

kind: Pod

metadata:

  name: nginxresourcheck

```
spec:

  containers:

  - image: nginx

    name: nginx
```

Once the Pod is created declaratively, using it will throw the error message:

Refer to the following Figure which shows the error message:

```
kishore@ss-MacBook-Pro chapter9 % kubectl create -f resourcecheck.yaml --namespace=resource-test
Error from server (Forbidden): error when creating "resourcecheck.yaml": pods "nginxresourcheck" is forbidden: failed quota: res-q
uota: must specify limits.cpu for: nginx; limits.memory for: nginx; requests.cpu for: nginx; requests.memory for: nginx
```

Figure 9.6: Resource quota error

This is because we have defined minimum and maximum resource requirements for objects in the resource quota for this namespace. We must ensure that the YAML manifest that creates the Pod defines these minimum and maximum resources.

Let us update the YAML definition of the Pod and recreate the resource Quota:

```
apiVersion: v1
```

```
kind: Pod

metadata:

  name: nginxresourcheck

spec:

  containers:

  - image: nginx

    name: nginx

    resources:

requests:

        cpu: “0.5”

        memory: “512m”


      limits:

        cpu: “1”

        memory: “1024m”
```

Create the Pod based on the above manifest file, which has the limit and request for cpu and the memory specified. And as the specified resource is within the limit of the resource quota specified, we can successfully create the Pod. Once the Pod is created and you query the resource quota from the following command:

$ kubectl describe resourcequota res-quota --namespace=resource-test

The above command describes the current state of the resource.

Refer to the following Figure which shows the current state of the resource quota:

```
kishore@ss-MacBook-Pro chapter9 % kubectl describe resourcequota res-quota --namespace=resource-test
Name:           res-quota
Namespace:      resource-test
Resource        Used   Hard
--------        ----   ----
limits.cpu      1      2
limits.memory   1024m  2096m
pods            1      1
requests.cpu    500m   1
requests.memory 1024m  1024m
```

Figure 9.7: Resource quota after Pod creation

Service account

Every action in a Kubernetes cluster needs to be authenticated and authorized. Service accounts are used for basic authentication from within the Kubernetes cluster. Every single Pod is using its service account.

If the service account is not specified, Pod uses its default service account. You can use the kubectl get service account command to query for the available service accounts.

As shown in Figure you should only see the default service account:

```
kishore@ss-MacBook-Pro chapter9 % kubectl get serviceaccounts
NAME      SECRETS   AGE
default   0         62d
kishore@ss-MacBook-Pro chapter9 %
```

Figure 9.8: Default service account display

You can find the service account of the Pod by querying the spec section:

$kubectl get pod nginx -o yaml | grep service

The above command will show the service account associated with the Pods. As shown in Figure the default service account is associated with the Pod:

```
kishore@ss-MacBook-Pro chapter9 % kubectl get po
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          6m9s
kishore@ss-MacBook-Pro chapter9 % kubectl get pod nginx -o yaml | grep service
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
  serviceAccount: default
  serviceAccountName: default
      - serviceAccountToken:
```

Figure 9.9: Service account associated with Pods

You can also create a service account and assign it to a Pod. We will see how:

The first step is to create a service account:

$kubectl create serviceaccount custom

The above command creates the service account custom.

As shown in Figure a service account custom is created:

```
kishore@ss-MacBook-Pro chapter9 % kubectl create serviceaccount custom
serviceaccount/custom created
kishore@ss-MacBook-Pro chapter9 %
```

Figure account is created

The next step is to associate the service account with the Pod:

You can define the service account as declarative. Below is the manifest file which creates the service account declarative. The note service account is

created as part of the

```
apiVersion: v1

kind: Pod

metadata:


  name: nginxsa

spec:

  serviceAccount: custom

  containers:

  - image: nginx:1.18.0

    name: nginx
```

As shown in Figure the service account custom is associated with Pod nginxsa:

```
kishore@ss-MacBook-Pro chapter9 % kubectl create -f sa.yaml
pod/nginxsa created
kishore@ss-MacBook-Pro chapter9 % kubectl describe po | grep -i service
Service Account:  custom
```

Figure 9.11: Custom service account assigned to the Pod

Conclusion

In this chapter, we looked at security contexts, privileges, and access control settings and learned about resource boundaries, how to set resource quota, and how to set a service account to a Pod.

In the next chapter, we will look into the Probes used by Kubernetes.

Questions

Define a security context on the container level of a new nginx Pod named security-demo. The security context adds the Linux capability CAP_SYS_TIME to the container.

Define a resource quota for the namespace The rules should constrain the count of configMap objects within the namespace to 1.

Create a new service account named Custom and assign it to a new Pod with an image nginx.

Solutions

Solution for question 1:

For the solution, we define the Alpine Pod named security-demo declaratively and add the Linux capability CAP_SYS_TIME to the container:

```
apiVersion: v1

kind: Pod

metadata:

  name: security-demo

spec:

  containers:

  - image: nginx

    name: security-demo

    securityContext:
```

```
    capabilities:

      add: ["SYS_TIME"]

  restartPolicy: Never
```

The next step is to create the Pod using the kubectl create command.

Solution for question 2:

First, create a namespace quotacheck:

$kubectl create namespace quotacheck

Create a YAML manifest for the resource quota with a hard value of 1 set for ConfigMap and save the file as

```
apiVersion: v1

kind: ResourceQuota

metadata:

  name: resourcequota

spec:
```

```
  hard:
    configMap: 1
```

Create the resource quota object in namespace quota check:

$kubectl create -f resource.yaml --namespace= quotacheck

Solution for question Q3:

Create a service account named custom using the imperative command:

$kubectl create serviceaccount custom

Create a manifest YAML file for creating Pod and associate it with service account:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginxsa
spec:
```

```
  serviceAccount: custom

  containers:

  - image: nginx:1.18.0

    name: nginx
```

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 10

Liveness and Readiness

Introduction

This chapter will discuss the various probes, namely readiness, liveness, and startup probes, the difference between liveness and readiness probes, and under which scenario to use. We will also learn how to configure them and debug scenarios associated with liveness and readiness probes.

Structure

In this chapter, we will cover the following topics:

Health probing

Readiness probe

Liveness probe

Startup probe

Objectives

The objective of this chapter is to ensure you have a good understanding of probes, namely readiness, liveness, and startup probes, and how to debug scenarios associated with liveness and readiness probes. So, let us get started.

Health probing

In today's fast-paced software development world, software bugs are inevitable. How will you ensure that issues like deadlocks, memory leaks, and infinite loops deployed in Pods and Kubernetes clusters are being mitigated and handled? The answer lies in Kubernetes health Probe.

A probe is a small test to determine if your Kubernetes Pod is still running, and this is quite useful in being proactive and preventing production incidents. Probe is a part of the Pod specification. Kubernetes has different probes flavors: Readiness probe, liveness probe, and startup probe. The readiness probe ensures that the Pod is ready to serve incoming requests. The liveness probe is used to check the availability of a Pod and check for the application's responsiveness and restarts the Pod if the container is in an unhealthy state. The startup probe is mostly used for legacy applications. Legacy applications require additional startup time, so the startup probe can give these legacy applications a little more time before the liveness probe can start probing.

Readiness probe

In Kubernetes readiness probe, as the name suggests, is when the container in the Pod is ready to accept traffic and requests. So, what defines a Pod as ready, well a Pod is ready when all its containers are ready to accept the traffic. Even when applications running on the Pods are started up, they will still require other configuration steps to be completed before it is completely ready. For example, it could be connecting to a database or making a service API call, so the readiness probe is used to ensure that all these steps are completed and Pod is fully ready to serve the requests. The readiness probe is invoked periodically and is used to determine whether a particular Pod is ready to accept the request. If the readiness probe is failed, it is removed from the service and is again added when the readiness probe is a success. How is this useful? Removing the failed probe will ensure that only healthy Pods serve client requests and clients using the application do not see the impact. Kubernetes service we will be looking at in detail in Chapter Kubernetes Networking and

The configuration of the readiness probe is as mentioned below:

```
readinessProbe:
  httpGet:
    path: /check
    port: 9090
```

initialDelaySeconds: 10

periodSeconds: 10

failureThreshold: 1

The readiness probe is added at the container level. From the above example, the HTTP request check is done at port and the context If the response is successful, httpGet is passed, and the container is ready to serve the request; if it fails, the container will not serve any requests.

initialDelaySeconds is the number of seconds to wait till the probe is initiated. For example, if you think the app running in the container will take around 10 seconds to start, then keep the value as 10 so that the probe starts after the start of the container.

periodSeconds is the value in seconds and defines how often the probes must be performed.

failureTreshold is the maximum number of consecutive health check failures before the container is considered unhealthy.

Let us look at one more example:

apiVersion: v1

kind: Pod

metadata:

```
  name: readiness-pod
spec:

  containers:
  - image: nginx
    name: readyme
    ports:
    -  containerPort: 8080
    readinessProbe:
      httpGet:
        path: /
        port: 8081
      initialDelaySeconds: 2
      periodSeconds: 8
```

In the above Pod definition YAML, you will see that the nginx is listening at port 8080, but the readiness probe is checking at 8081. For the above Pod definition YAML which has a readiness probe, creates the pod by pointing the create command to the YAML manifest.

$kubectl create -f readinesspod.yml

The above command will create the Pod from YAML.

As the readiness probe is checking at port 8081, and the container is listening at Pod 8080, the probe will fail, and the container will never be ready; that is, it is not ready to accept any request.

Refer to Figure which shows how the Pod is not ready (0/1), and it is due to readiness probe failure:

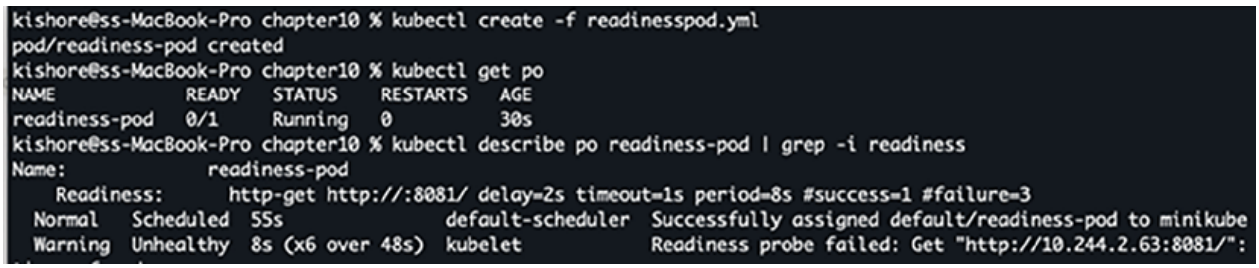

kishore@ss-MacBook-Pro chapter10 % kubectl create -f readinesspod.yml
pod/readiness-pod created
kishore@ss-MacBook-Pro chapter10 % kubectl get po
NAME READY STATUS RESTARTS AGE
readiness-pod 0/1 Running 0 30s
kishore@ss-MacBook-Pro chapter10 % kubectl describe po readiness-pod | grep -i readiness
Name: readiness-pod
Readiness: http-get http://:8081/ delay=2s timeout=1s period=8s #success=1 #failure=3
Normal Scheduled 55s default-scheduler Successfully assigned default/readiness-pod to minikube
Warning Unhealthy 8s (x6 over 48s) kubelet Readiness probe failed: Get "http://10.244.2.63:8081/":

Figure 10.1: Readiness probe details

If you change the readiness probe to 8080 in the Pod definition YAML file, which is the same port as where the container is listening, and try to create the pod, you will see the readiness probe is successful, and the Pod is in a ready state (1/1).

In the above example, we saw the httpGet probe, which sends the HTTP request to the container, and the HTTP status of the container tells if the container is healthy or not.

There are two other probes:

Exec probes where the process is executed, and the process exit status determines the container’s status.

TCP socket probe opens the TCP connection to the container port, and if the connection is established, it is considered a success.

Liveness probe

So, what is the main difference between the liveness and readiness probe? In the liveness probe, if the probe is failed, the container will be killed and restarted.

The liveness probe is a way for Kubernetes to know when to restart the container. For example, if the application is not responding due to deadlock, liveness could catch it and restart the container, which usually helps to clear the deadlock, and the application responds.

Let us look at this in detail with an example:

```
apiVersion: v1

kind: Pod

metadata:

  name: livenesscheck

spec:

  containers:

  - image: busybox
```

```
    name: app

    args:

    - /bin/sh

    - -c



    - while true; do touch /tmp/test.txt; sleep 36; rm -f /tmp/test.txt; sleep 600;
done;

    livenessProbe:

      exec:

        command:

        - cat /temp/test.txt

      initialDelaySeconds: 5

      periodSeconds: 5
```

In the above example, we have defined a container app that creates a file and after 36 seconds, it deletes the file. The liveness probes use probe type exec and check for the test.txt every 5 seconds. When the above example is executed, it will be the first time the liveness probe will be successful, as the

/tmp/test.txt file exists. However, after removing the file, the subsequent liveness probe check will fail after 5 seconds.

And this will cause the Pod to restart.

$kubectl describe po | grep Events -A 20

The above command helps print the events of the Pod, which will describe how many times the Pod has been restarted.

$ kubectl get events --field-selector involvedObject.name=”livenesscheck”

The above command displays the events of the Pod.

Refer to Figure which shows how Pod is created:

kishore@ss-MacBook-Pro chapter10 % kubectl describe po | grep Events -A 20
Events:
Type Reason Age From Message
Normal Scheduled 55s default-scheduler Successfully assigned default/livenesscheck to minikube
Normal Pulling 54s kubelet Pulling image "busybox"
Normal Pulled 49s kubelet Successfully pulled image "busybox" in 5.0660053s (5.0660371s ir
Normal Created 49s kubelet Created container app
Normal Started 49s kubelet Started container app

Figure 10.2: Liveness probe Pod creation

Refer to Figure which shows how the liveness probe takes into effect and causes the restart of the Pod:

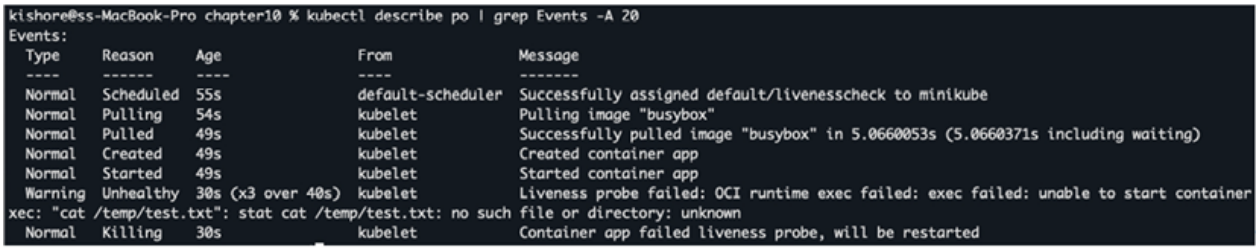

kishore@ss-MacBook-Pro chapter10 % kubectl describe po | grep Events -A 20
Events:
Type Reason Age From Message
Normal Scheduled 55s default-scheduler Successfully assigned default/livenesscheck to minikube
Normal Pulling 54s kubelet Pulling image "busybox"
Normal Pulled 49s kubelet Successfully pulled image "busybox" in 5.0660053s (5.0660371s including waiting)
Normal Created 49s kubelet Created container app
Normal Started 49s kubelet Started container app
Warning Unhealthy 30s (x3 over 40s) kubelet Liveness probe failed: OCI runtime exec failed: exec failed: unable to start container
xec: "cat /temp/test.txt": stat cat /temp/test.txt: no such file or directory: unknown
Normal Killing 30s kubelet Container app failed liveness probe, will be restarted

Figure 10.3: Liveness probe details

Like readiness probes, all three exec probe, and TCP socket probes exist for the liveness probes.

Also, the configuration parameters and failureThreshold for the liveness probe are the same as readiness. Below are some additional configuration parameters:

Defines the time-out period of the probe. For example, if the probe is httpGet and the timeoutseconds is set to 2 seconds, then the probe is considered a failure if there is no response within 3 seconds.

The minimum number of consecutive health checks before the probe is successful.

Startup probe

Startup probes are mostly used for legacy applications, which take time to start completely. So, the readiness and liveness probe will not be effective unless a startup probe is complete.

Again, as with other probes, let us look at this with an example:

```
apiVersion: v1

kind: Pod

metadata:

  name: startupcheck

spec:

  containers:

  - image: httpd:2.4.46

    name: http-server

    startupProbe:
```

tcpSocket:

port: 80

initialDelaySeconds: 3

periodSeconds: 15

Refer to Figure which shows how Pod is created:

```
kishore@ss-MacBook-Pro chapter10 % kubectl create -f startup.yaml
pod/startupcheck created
kishore@ss-MacBook-Pro chapter10 % kubectl get po
NAME           READY   STATUS              RESTARTS   AGE
startupcheck   0/1     ContainerCreating   0          9s
kishore@ss-MacBook-Pro chapter10 % kubectl get po
NAME           READY   STATUS    RESTARTS   AGE
startupcheck   1/1     Running   0          19s
```

Figure 10.4: Startup probe pod creation

Refer to Figure which shows how the startup probe takes into effect:

```
kishore@ss-MacBook-Pro chapter10 % kubectl describe po startupcheck | grep Events -A 20
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  41s   default-scheduler  Successfully assigned default/startupcheck to minikube
  Normal  Pulling    41s   kubelet            Pulling image "httpd:2.4.46"
  Normal  Pulled     32s   kubelet            Successfully pulled image "httpd:2.4.46" in 9.4571676s (9.457239s including waiting)
  Normal  Created    31s   kubelet            Created container http-server
  Normal  Started    31s   kubelet            Started container http-server
```

Figure 10.5: Startup probe details

Conclusion

In this chapter, we looked at different flavors of Kubernetes probes, namely the readiness probe, the liveness probe, and the start-up probe.

The next chapter will look into monitoring and debugging K8s Clusters.

Questions

Create an nginx Pod named check-probe. Add probes to the Pod so that if the application inside is not responding to HTTP requests on port 8080, it should be restarted.

Show all the events associated with the Pod check probe.

Create an nginx image Pod named probe test. Add the following arguments to the Pod. Create a probe that checks if a file on that path /tmp/myfile exists. If it does not exist, the Pod should be restarted:

- /bin/sh

- -c

- touch /tmp/myfile; sleep 3600

Solutions

Solution to question 1:

For the solution, we define the nginx Pod named check-probe declaratively and use the kubectl create command to create Pod with probe. We need to use the HTTP method for liveness probe since a restart is required:

```
apiVersion: v1

kind: Pod

metadata:

  name: check-probe

spec:

  containers:

  - image: nginx

    name: check-probe

    ports:
```

```
      containerPort: 8080
    readinessProbe:
      httpGet:
        path: /

        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 5
```

Solution to question 2:

You can use the following command to get the event details of the Pod

```
kubectl get events --field-selector involvedObject.name=”check-probe”
```

Solution to question 3:

Create the Pod based on the below YAML manifest. Use the kubectl create command:

```
apiVersion: v1
```

```
kind: Pod

metadata:

  name: probetest

spec:

  containers:

  - image: nginx

    name: probetest



    args:

    - /bin/sh

    - -c

    - touch /tmp/myfile; sleep 3600

    livenessProbe:

      exec:
```

```
      command:

      - cat /temp/myfile

    initialDelaySeconds: 3

    periodSeconds: 3
```

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 11

Monitoring and Debugging of Kubernetes Cluster

Introduction

This chapter will discuss how to monitor and debug the Kubernetes cluster. You will learn strategies for debugging a misconfigured or misbehaving Kubernetes object. Also, we will see how to monitor clusters and nodes.

Structure

In this chapter, we will cover the following topics:

Debugging Kubernetes cluster and strategies

Monitor Kubernetes cluster and Pods.

Monitoring events and logs

Observability and monitoring

Debugging and monitoring tips and tricks

Objectives

The objective of this chapter is to ensure you have a good understanding of how to monitor and debug the Kubernetes cluster, which includes Pods and Nodes. So, let us get started.

Debugging Kubernetes cluster and strategies

For this book, we have been using a minikube cluster to create Kubernetes components, and we have 5-10 Kubernetes objects running simultaneously. But the real-life Kubernetes production environment is quite different; hundreds of Kubernetes objects will be running together, serving the end user's request. In this scenario, where hundreds of components are running together, you will inevitably face failure. So, if any failure occurs, you should know how to troubleshoot. You should have some strategies in place as to which components you should be looking at and how to troubleshoot. The methods discussed in subsequent sections start with a high level and drill into detail as needed.

Monitor Kubernetes cluster and Pods

Let us first start with the application or container logging, and let us begin with an example. Consider the below YAML file. The Pod definition YAML file has a container named ping whose job is to ping

```
apiVersion: v1

kind: Pod

metadata:

  name: ping

spec:

  containers:

  - image: busybox

    name: pingmain

    command: [“ping”]

    args: [“kubernetes.io”]
```

Use the kubectl create command to create the Pod. And once the Pod is running, use the kubectl logs name> to check the application logs.

The following command logs from the application Pod:

$ kubectl logs ping

Refer to Figure which shows how the application Pod is logging application data which can be helpful for troubleshooting purposes, the IP address shown in the following figure is the IP address of

```
kishore@ss-MacBook-Pro chapter11 % kubectl create -f ping.yaml
pod/ping created
kishore@ss-MacBook-Pro chapter11 % kubectl get po
NAME   READY   STATUS    RESTARTS   AGE
ping   1/1     Running   0          8s
kishore@ss-MacBook-Pro chapter11 % kubectl logs ping
PING kubernetes.io (147.75.40.148): 56 data bytes
64 bytes from 147.75.40.148: seq=0 ttl=36 time=4.354 ms
64 bytes from 147.75.40.148: seq=1 ttl=36 time=10.989 ms
64 bytes from 147.75.40.148: seq=2 ttl=36 time=15.698 ms
64 bytes from 147.75.40.148: seq=3 ttl=36 time=11.304 ms
64 bytes from 147.75.40.148: seq=4 ttl=36 time=12.392 ms
64 bytes from 147.75.40.148: seq=5 ttl=36 time=5.773 ms
64 bytes from 147.75.40.148: seq=6 ttl=36 time=10.334 ms
64 bytes from 147.75.40.148: seq=7 ttl=36 time=11.084 ms
64 bytes from 147.75.40.148: seq=8 ttl=36 time=12.819 ms
64 bytes from 147.75.40.148: seq=9 ttl=36 time=11.241 ms
64 bytes from 147.75.40.148: seq=10 ttl=36 time=5.577 ms
64 bytes from 147.75.40.148: seq=11 ttl=36 time=17.938 ms
64 bytes from 147.75.40.148: seq=12 ttl=36 time=11.967 ms
64 bytes from 147.75.40.148: seq=13 ttl=36 time=10.876 ms
64 bytes from 147.75.40.148: seq=14 ttl=36 time=6.819 ms
```

Figure 11.1: Checking and Logging application log

For the scenarios where you have more than one container in the Pod and you want a container-specific log, you can use the same log option but add the container name. Let us see it with an example.

The exact ping example we used earlier is now modified to have the additional container. Below is the updated Pod YAML definition, which has two containers, one pingmain and the second one

```
apiVersion: v1

kind: Pod



metadata:

  name: pingtwo

spec:

  containers:

  - image: busybox

    name: pingmain

    command: [“ping”]

    args: [“kubernetes.io”]
```

```
- image: busybox

  name: pingsec

  command: [“ping”]

  args: [“kubernetes.io”]
```

Use the kubectl create command to create the Pod. And once the Pod is running, use the kubectl logs name> name> to check the application logs.

The following command logs from the application Pod and specific to container

```
$ kubectl logs pingtwo pingsec
```

Refer to Figure which shows how the application Pod is logging application data from a specific container, which can be helpful for troubleshooting purposes, the IP address shown in the figure is the IP address of

Note the container running is 2/2; that is, two containers are running in the Pod.

```
kishore@ss-MacBook-Pro chapter11 % kubectl create -f pingsec.yaml
pod/pingtwo created
kishore@ss-MacBook-Pro chapter11 % kubectl get po
NAME      READY   STATUS    RESTARTS   AGE
pingtwo   2/2     Running   0          11s
kishore@ss-MacBook-Pro chapter11 % kubectl logs pingtwo pingsec
PING kubernetes.io (147.75.40.148): 56 data bytes
64 bytes from 147.75.40.148: seq=0 ttl=36 time=12.549 ms
64 bytes from 147.75.40.148: seq=3 ttl=36 time=13.145 ms
64 bytes from 147.75.40.148: seq=1 ttl=36 time=5.871 ms
64 bytes from 147.75.40.148: seq=4 ttl=36 time=5.950 ms
64 bytes from 147.75.40.148: seq=2 ttl=36 time=9.652 ms
64 bytes from 147.75.40.148: seq=5 ttl=36 time=3.966 ms
64 bytes from 147.75.40.148: seq=3 ttl=36 time=6.463 ms
64 bytes from 147.75.40.148: seq=6 ttl=36 time=5.708 ms
64 bytes from 147.75.40.148: seq=4 ttl=36 time=10.270 ms
64 bytes from 147.75.40.148: seq=7 ttl=36 time=12.302 ms
64 bytes from 147.75.40.148: seq=5 ttl=36 time=6.662 ms
64 bytes from 147.75.40.148: seq=8 ttl=36 time=13.674 ms
64 bytes from 147.75.40.148: seq=6 ttl=36 time=5.640 ms
64 bytes from 147.75.40.148: seq=9 ttl=36 time=6.029 ms
64 bytes from 147.75.40.148: seq=7 ttl=36 time=5.337 ms
```

Figure 11.2: Logging application log from a specific container

There is one more option which gets the logs from the previous instantiation of a container. You can use this command to check the logs from the container that has been restarted.

Sometimes you see Pod creation command status success, but the container which is serving the Pod is not ready. For example, consider the below YAML manifest of Pod definition:

apiVersion: v1

kind: Pod

```
metadata:

  name: checkpod

spec:

  containers:

  - image: nginxx

    name: checkpod
```

We have deliberately added the wrong image name of If we create the Pod using the kubectl create command, you will see that the command is successful.

Refer to Figure which shows that Pod was successfully created:

Figure 11.3: Create Pod command executed successfully

If you do a get Pod command, you will see the container is not ready (as its 0/1 and not 1/1), and there is cause for this as the image was not found (From the status message and it is expected as the image is not a correct image name.

Refer to Figure which shows that there is an error while getting the image from the image repo:

```
kishore@ss-MacBook-Pro chapter11 % kubectl get po
NAME       READY   STATUS             RESTARTS   AGE
checkpod   0/1     ImagePullBackOff   0          42s
```

Figure 11.4: ImagePullBackoff error

In the above example, we have seen ImagePullBackOff as an error, but there are other common errors that you need to be aware of. The following table describes the additional common error:

Table 11.1: List of common errors

Monitor events and logs

As seen in the previous section, Monitor Kubernetes Cluster and the Pod status message provides only a high-level cause of the issue but no details and does not provide sufficient information to debug further.

Refer to Figure The status message only shows

```
kishore@ss-MacBook-Pro chapter11 % kubectl get po
NAME       READY   STATUS             RESTARTS   AGE
checkpod   0/1     ImagePullBackOff   0          42s
```

Figure 11.5: ImagePullBackoff error

If you want to know more details as to what has caused the issue, use describe Pod commands and look at the event section:

$ kubectl describe Pod checkpod

Refer to Figure The events section of the describe Pod command shows more details of the cause of the issue:

```
                 node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type     Reason     Age                From               Message
  ----     ------     ----               ----               -------
  Normal   Scheduled  19m                default-scheduler  Successfully assigned default/checkpod to minikube
  Normal   Pulling    17m (x4 over 19m)  kubelet            Pulling image "nginxx"
  Warning  Failed     17m (x4 over 19m)  kubelet            Failed to pull image "nginxx": rpc error: code = Unknown desc = Error resp
onse from daemon: pull access denied for nginxx, repository does not exist or may require 'docker login': denied: requested access to th
e resource is denied
```

Figure 11.6: Events section shows the detailed cause

You can also use the get events command. This command will show all the events across all the Pods for a given namespace.

The following command provides events details across all the Pods of the namespace:

$ kubectl get events

Refer to Figure which shows events across all the Pods in the namespace:

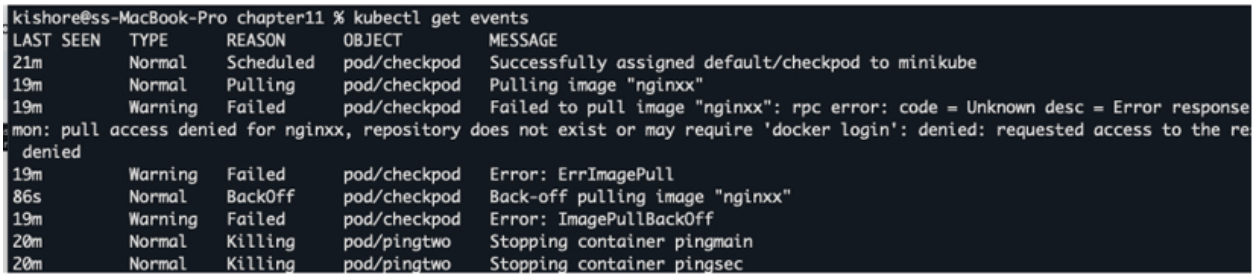

kishore@ss-MacBook-Pro chapter11 % kubectl get events
LAST SEEN TYPE REASON OBJECT MESSAGE
21m Normal Scheduled pod/checkpod Successfully assigned default/checkpod to minikube
19m Normal Pulling pod/checkpod Pulling image "nginxx"
19m Warning Failed pod/checkpod Failed to pull image "nginxx": rpc error: code = Unknown desc = Error response
mon: pull access denied for nginxx, repository does not exist or may require 'docker login': denied: requested access to the re
denied
19m Warning Failed pod/checkpod Error: ErrImagePull
86s Normal BackOff pod/checkpod Back-off pulling image "nginxx"
19m Warning Failed pod/checkpod Error: ImagePullBackOff
20m Normal Killing pod/pingtwo Stopping container pingmain
20m Normal Killing pod/pingtwo Stopping container pingsec

Figure 11.7: Shows event detail across all the Pods in the namespace

To monitor debug, Kubernetes has a field selector option. The field selector lets you select the Kubernetes resources based on the value of one or more resource fields.

For example, say you want to get all the Pods to be displayed except the Pod in the default namespace.

The following command will display Pod from all the namespaces other than the default namespace:

$kubectl get pods --all-namespaces --field-selector metadata.namespace!=default

Refer to Figure Pods from all the namespace other than the default namespace are displayed:

```
kishore@ss-MacBook-Pro chapter11 % kubectl get pods --all-namespaces --field-selector metadata.namespace!=default
NAMESPACE      NAME                               READY   STATUS              RESTARTS         AGE
kube-system    coredns-787d4945fb-8tkq4           1/1     Running             16 (3h30m ago)   69d
kube-system    etcd-minikube                      1/1     Running             16 (3h30m ago)   69d
kube-system    kube-apiserver-minikube            1/1     Running             16 (3h30m ago)   69d
kube-system    kube-controller-manager-minikube   1/1     Running             16 (3h30m ago)   69d
kube-system    kube-proxy-l5dwl                   1/1     Running             16 (3h30m ago)   69d
kube-system    kube-scheduler-minikube            1/1     Running             16 (3h30m ago)   69d
kube-system    metrics-server-5f8fcc9bb7-jzlhs    1/1     Running             0                146m
kube-system    storage-provisioner                1/1     Running             30 (3h29m ago)   69d
resource-test  nginxresourcecheck                 0/1     ContainerCreating   0                6d23h
```

Figure 11.8: Pods from all the namespaces except default are displayed

The following command will display all Pod only from the default namespace:

$kubectl get pods --all-namespaces --field-selector metadata.namespace=default

Refer to Figure Pods from only the default namespace are displayed:

```
kishore@ss-MacBook-Pro chapter11 % kubectl get pods --all-namespaces --field-selector metadata.namespace=default
NAMESPACE   NAME       READY   STATUS    RESTARTS   AGE
default     checkpod   1/1     Running   0          75s
```

Figure 11.9: Pods from only the default namespace are displayed

So, when you want to search particular objects in the Kubernetes cluster based on certain constraints, you can use the field selector.

Let us look at one more example. Say you want to get information on a particular Pod. You can use field selector,

The following command shows the use of a field selector to get information about Pod:

$kubectl get pods --field-selector metadata.name=checkpod

Refer to Figure the field selector used to get Pod information:

```
kishore@ss-MacBook-Pro chapter11 % kubectl get pods --field-selector metadata.name=checkpod
NAME       READY   STATUS    RESTARTS   AGE
checkpod   1/1     Running   0          11m
```

Figure 11.10: Use the field selector to get information on Pod

Now let us say you want to display all the events related to Pod checkpod. This can be achieved using a field selector.

You can use the kubectl get events command and output it to

$kubectl get events -o json

The above command gets events and puts them in json format.

Refer to Figure the events are output to json format:

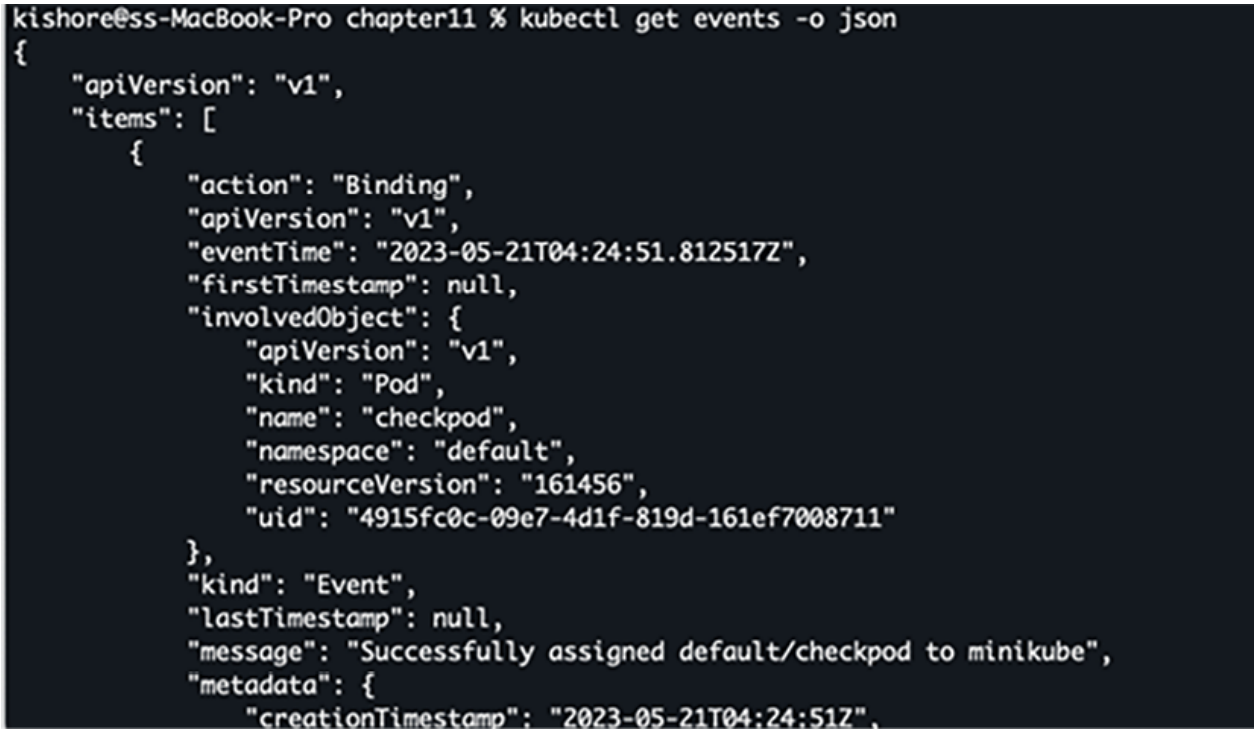


Figure 11.11: Event output in json format

If you look at a Json object, there is an involved object section that tells you the kind and the name of the object, in our case kind is Pod, and the name is

“involvedObject”: {

“apiVersion”: “v1”,

“kind”: “Pod”,

“name”: “checkpod”,

“namespace”: “default”,

“resourceVersion”: “161456”,

"uid": "4915fc0c-09e7-4d1f-819d-161ef7008711"

},

If you want to get events related to a specific Pod, you can use a field selector and use an involved object, in our case involvedobject and kind equal to Pod.

The following command is to get events associated with Pod using the field selector:

$kubectl get events --field-selector involvedObject.kind=Pod

Refer to Figure to get the events associated with Pods:

```
kishore@ss-MacBook-Pro chapter11 % kubectl get events --field-selector involvedObject.kind=Pod
LAST SEEN   TYPE      REASON      OBJECT          MESSAGE
58m         Normal    Scheduled   pod/checkpod    Successfully assigned default/checkpod to minikube
57m         Normal    Pulling     pod/checkpod    Pulling image "nginxx"
57m         Warning   Failed      pod/checkpod    Failed to pull image "nginxx": rpc error: code = Unknown desc = Error response from dae
mon: pull access denied for nginxx, repository does not exist or may require 'docker login': denied: requested access to the resource is
 denied
57m         Warning   Failed      pod/checkpod    Error: ErrImagePull
23m         Normal    BackOff     pod/checkpod    Back-off pulling image "nginxx"
57m         Warning   Failed      pod/checkpod    Error: ImagePullBackOff
21m         Normal    Scheduled   pod/checkpod    Successfully assigned default/checkpod to minikube
21m         Normal    Pulling     pod/checkpod    Pulling image "nginx"
21m         Normal    Pulled      pod/checkpod    Successfully pulled image "nginx" in 2.4564464s (2.4564945s including waiting)
21m         Normal    Created     pod/checkpod    Created container checkpod
21m         Normal    Started     pod/checkpod    Started container checkpod
58m         Normal    Killing     pod/pingtwo     Stopping container pingmain
58m         Normal    Killing     pod/pingtwo     Stopping container pingsec
```

Figure 11.12: Display events associated with Pods

Observability and monitoring

Another aspect of motoring that developers should know is how the Pod consumes resources. This resource information is needed to plan out the optimal resources the application will need. In the real world, you have various monitoring tools like Prometheus, which help collect, process, and visualize the data from the Kubernetes cluster.

In the CKAD exam, you are not expected to know commercial tools like Prometheus. The exam relies on the metric server, which aggregates the data of resource usage across the Kubernetes cluster. This metric cluster collects the underlying use of resources like CPU and memory.

The first step is to add the metric server to the minikube:

$ minikube addons enable metrics-server

The above command will enable the metric server.

Refer to Figure which shows the metric server is enabled to cluster:

kishore@ss-MacBook-Pro chapter11 % minikube addons enable metrics-server
metrics-server is an addon maintained by Kubernetes. For any concerns contact minikube on GitHub.
You can view the list of minikube maintainers at: https://github.com/kubernetes/minikube/blob/master/OWNERS
Using image registry.k8s.io/metrics-server/metrics-server:v0.6.2
The 'metrics-server' addon is enabled

Figure 11.13: Metric server enabled

Once the metric server is enabled, you can use the top command to query the resources of nodes and Pods.

The following command displays a snapshot of near-real-time resource usage (CPU and Memory) of nodes in the cluster:

$ Kubectl top nodes

Refer to Figure which shows the resource usage of nodes:

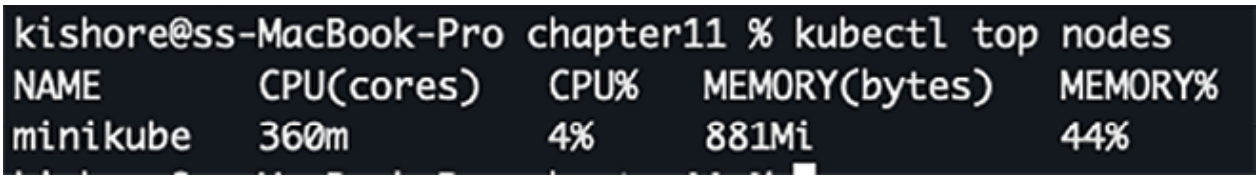


Figure 11.14: Resource usage of nodes

Likewise, you can query the resource utilization of Pod using the same top command.

The following command displays a snapshot of near-real-time resource usage (CPU and Memory) of Pods in the cluster.

$kubectl top Pod

Refer to Figure which shows the resource usage of Pods:

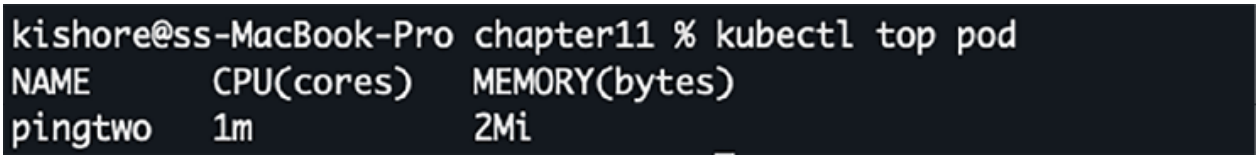


Figure 11.15: Resource usage of Pod

If you want to explicitly check the resource usage of Pod, just append the Pod name to the Top command.

Refer to Figure which shows the resource usage of specific Pod

```
kishore@ss-MacBook-Pro chapter11 % kubectl top pod pingtwo
NAME      CPU(cores)   MEMORY(bytes)
pingtwo   1m           2Mi
```

Figure 11.16: Resource usage of specific Pod

Debugging and monitoring tips and tricks

Debugging and monitoring are important topics both from the CKAD exam point of view and to solve real-world Kubernetes issues. Below are some points you can keep in mind during debugging and monitoring the Kubernetes cluster:

Always do logs command to see what is going on inside the application running on Kubernetes.

Use the describe Pod commands to check the various stages of Pod and identify the cause of failure.

Keep an eye on the ready status sometimes, the Pod may be in the running status, but the container may not be up to serve the request. Check the container logs to see the cause of the issue.

Use the gets events command to do a deep dive into the events.

Use the Top command to inspect the metric usage, that is, usage of CPU and memory both at Pod and node.

Conclusion

In this chapter, we looked how at various strategies to monitor and debug the Kubernetes cluster. This includes monitor of nodes and Pods. In the next chapter, we will look into networking and services.

Questions

Apply the below manifest file in your Kubernetes cluster. Wait for the next 30 seconds. Observe that the pod named monitor keeps on restarting. Check and fix the issue:

```
apiVersion: v1

kind: Pod

metadata:

  name: monitor

spec:

  containers:

  - image: nginx

    name: monitor

    livenessProbe:

    httpGet:
```

```
    path: /

    port: 8080

  initialDelaySeconds: 3

  periodSeconds: 3
```

Store all the events associated with the Pod monitor and store them to /tmp/podlog.txt

Inspect the metric details of the Pod. Find the CPU and memory associated with the Pod monitor and store it in

In the format CPU: < value> . Memory

Solutions

Solution to question 1:

Create the Pod using the command kubectl create command:

$ kubectl create -f faultpod.yaml

The monitoring Pod keeps on restarting, and the nginx image runs on Port 80, but the liveness probe is configured on Port 8080. To fix it, edit the configuration (The YAML file) and change the Port from 8080 to 80. Delete the Pod and recreate the Pod again.

Solution to question 2:

You can use the field selector to get events associated with the Pod monitor and store them in the file

$ kubectl get events --field-selector involvedObject.name=”monitor” > /tmp/podlog.txt

Solution to question 3:

Use the kubectl top command to get the metrics of the Pod monitor:

$ kubectl top pod monitor

Copy the CPU and memory and store it in /tmp/monitor.txt in the format CPU : < value> . Memory

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 12

Kubernetes Networking and Services

Introduction

In this chapter, we will discuss various types of Kubernetes Services, and you will also learn about Kubernetes networking and network policy. So, let us get started.

Structure

In this chapter, we will cover the following topics:

Kubernetes Services and types

Creating Services

Deployment and Services

Kubernetes networking and network policy

Troubleshooting Service

Objectives

The objective of this chapter is to ensure you have a good understanding of Kubernetes Services and its various declared types and get a good understanding of Kubernetes networking and network policy.

Kubernetes Services and types

Pods are ephemeral in nature and prone to frequent restarts. When Pod restarts, the Pod IP changes. Hence Pod does not have constant IPs. When a Pod is restarted, which will be frequently, a new IP is assigned to the Pod. So, you cannot build a communication protocol where end users or part of the system rely on Pod IP to communicate with Pod. Kubernetes Service is used to front these, Pods. Kubernetes Service has fixed virtual IPs, which do not change.

The Service object in Kubernetes exposes a logical set of Pods (which in turn host application) in the network. By default, the Service uses round-robin load balancing to forward traffic to specific Pods. There is no physical load balancer, and this load balancer is configured at the Kubernetes level.

A selector determines the Pods Service targets, and the selector is a label. We had seen how the label is used as a selector in Chapter Pod Design and Service acts as an independent entity and is not deleted when deployment or Pods is deleted from the cluster. The only thing that the Service is doing is that it watches for Pods that have a specific label set, which matches the selector that is specified in that Service.

Kubernetes provides three Service flavors: ClusterIP, NodePort, and LoadBalancer. Let us look at these types:

They are the default Service type. Cluster IP exposes the Service on an internal Kubernetes cluster, and this IP is not exposed to external users that is, this IP is only reachable within the Kubernetes cluster.

In NodePort, port forwarding is added, a high-level port is dynamically allocated, and that is accessible by external users. The dynamically allocated port is accessed by external users to reach actual end nodes. So, in a way, NodePortIP is accessed by external users.

Exposes the Services externally using a cloud load balancer. This is currently only implemented in the public cloud.

For your CKAD exam, you should focus on ClusterIP and NodePort.

Creating Services

So let us start by creating a Service that can be created imperatively and declaratively, like other Kubernetes objects.

First, let us look at the imperative way. Kubectl create service command, creates a Service and the default type is In this command, you can also mention port forwarding.

$kubectl create service clusterip service-nginx --tcp=80:80

The above command creates a Service named of default type

Refer to Figure which shows an imperative way to create a Service:

```
kishore@ss-MacBook-Pro chapter12 % kubectl create service clusterip service-nginx --tcp=80:80
service/service-nginx created
```

Figure 12.1: Imperative way to create Service

You can use the kubectl get commands to get the Service in the cluster.

$kubectl get service.

The above command defines the Service in the cluster.

Refer to Figure which shows the kubectl command to get Services from the cluster:

```
kishore@ss-MacBook-Pro chapter12 % kubectl get service service-nginx
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
service-nginx   ClusterIP   10.104.233.69   <none>        80/TCP    3m30s
```

Figure 12.2: Kubectl get command to get the Service

In the previous example, we saw how to create a Service as a stand-alone object, but the Service can be created as part of creating a Pod or deployment. For this, you need to use the expose command. We will see this in the following example:

$kubectl run nginxpod --image=nginx --restart=Never --port=80 –-expose

The above command creates an nginx pod with the name also, the arguments create a corresponding Service with the correct label for selection.

Refer to Figure which shows how to expose arguments are used to create Pod and its corresponding Service:

```
kishore@ss-MacBook-Pro chapter12 % kubectl run nginxpod --image=nginx --restart=Never --port=80 --expose
service/nginxpod created
pod/nginxpod created
```

Figure 12.3: kubectl expose command to create Pod and its Service

You can also use expose deployment to create a Service for an existing deployment.

Refer to Figure which shows the existing deployment of my-dep existing in the cluster:

```
kishore@ss-MacBook-Pro chapter12 % kubectl get deployment
NAME     READY   UP-TO-DATE   AVAILABLE   AGE
my-dep   1/1     1            1           7s
```

Figure 12.4: Existing deployment in the cluster

You can create a Service to existing deployment

$kubectl expose deployment my-dep --port=80 --target-port=80

The above command creates a Service for deployment

Refer to Figure which shows how to expose argument is used to create Service for deployment imperative:

```
kishore@ss-MacBook-Pro chapter12 % kubectl expose deployment my-dep --port=80 --target-port=80
service/my-dep exposed
```

Figure 12.5: Service for deployment created

The Service can be created declaratively using the YAML file as with other Kubernetes objects. Refer to the following example of the YAML definition file for Service:

apiVersion: v1

kind: Service

```
metadata:
  name: service-nginx
spec:
  type: ClusterIP
  selector:
    app: service-nginx
  ports:
  - port: 80
    targetPort: 80
```

Like other objects, we can use describe command to get more information about the Service object

```
$kubectl describe service my-dep
```

The above command describes the details of the service my-dep, which was created earlier.

Refer to Figure which shows the describe service kubectl command. Note by default, the Service type is

```
kishore@ss-MacBook-Pro chapter12 % kubectl describe service my-dep
Name:              my-dep
Namespace:         default
Labels:            app=my-dep
Annotations:       <none>
Selector:          app=my-dep
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.106.77.96
IPs:               10.106.77.96
Port:              <unset>  80/TCP
TargetPort:        80/TCP
Endpoints:         10.244.2.84:80
Session Affinity:  None
Events:            <none>
```

Figure 12.6: Description of the Service object

Let us try to explore the cluster IP using an example. As mentioned before, if the Service type is not mentioned, the default Service type created is the type So, considering the previous example, a ClusterIP Service my-dep was created for deployment ClusterIP is exposed within the cluster, so we can create a Pod in the same cluster and see if we can use the wget command to access the application via Service

From the above Figure you can see the ClusterIP is exposed at IP 10.106.77.96 port 80.

$kubectl run busybox --image=busybox --restart=Never -it -- /bin/sh

The above command starts a busybox Pod. Once the Pod is up, access the my-dep Service.

Refer to Figure which shows how to access the application via the wget command:

```
kishore@ss-MacBook-Pro chapter12 % kubectl run busybox --image=busybox --restart=Never -it -- /bin/sh
If you don't see a command prompt, try pressing enter.
/ #
/ #
/ # wget -O- 10.106.77.96:80
Connecting to 10.106.77.96:80 (10.106.77.96:80)
writing to stdout
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
```

Figure 12.7: Access the ClusterIP Service

When you use create NodePort Service, then the Service is accessible outside the cluster. The node's IP address is assigned automatically upon the creation of the Service and can be reached in combination with a port number in the range of 30000 and 32767.

For the NodePort example, let us use the same example as the previous my-dep Service, defined as The first step is to edit the service my-dep and make the type as For that, you can use the patch command:

$kubectl patch service my-dep -p '{ "spec": {"type": "NodePort"} }

Refer to Figure where the service my-dep is patched to NodePort

```
kishore@ss-MacBook-Pro chapter12 % kubectl patch service my-dep -p '{ "spec": {"type": "NodePort"} }'
service/my-dep patched
kishore@ss-MacBook-Pro chapter12 % kubectl get service my-dep
NAME     TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
my-dep   NodePort   10.106.77.96   <none>        80:31221/TCP   47m
```

Figure 12.8: ClusterIP patched to the NodePort

Refer to Figure which shows the details of NodeType service

```
kishore@ss-MacBook-Pro chapter12 % kubectl describe service my-dep
Name:                     my-dep
Namespace:                default
Labels:                   app=my-dep
Annotations:              <none>
Selector:                 app=my-dep
Type:                     NodePort
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.106.77.96
IPs:                      10.106.77.96
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  31221/TCP
Endpoints:                10.244.2.84:80
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

Figure 12.9: NodePort details

Once NodePort is created, the Service is exposed outside the cluster and can be called outside the Kubernetes Cluster.

Deployment and Service

We have touched upon this point earlier, the deployment and the Services work independently, deleting the deployment object does not delete the corresponding Service object. A deployment manages Pods and their replication, and a Service routes network requests to a set of Pods.

For example, the Service my-dep still exists if we delete the deployment my-dep from the previous section.

Refer to Figure which shows the Service exists even when the corresponding deployment is deleted:

```
kishore@ss-MacBook-Pro chapter12 % kubectl get deployment
NAME     READY   UP-TO-DATE   AVAILABLE   AGE
my-dep   1/1     1            1           76m
kishore@ss-MacBook-Pro chapter12 % kubectl get service my-dep
NAME     TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
my-dep   NodePort   10.106.77.96   <none>        80:31221/TCP   74m
kishore@ss-MacBook-Pro chapter12 % kubectl delete deployment my-dep
deployment.apps "my-dep" deleted
kishore@ss-MacBook-Pro chapter12 % kubectl get service my-dep
NAME     TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
my-dep   NodePort   10.106.77.96   <none>        80:31221/TCP   74m
```

Figure 12.10: Service exists even if deployment is deleted

Kubernetes networking and network policy

Within a Kubernetes cluster, Pods can communicate with other Pods using IP or DNS name. These communications can also happen across namespaces. There are scenarios where you want to restrict these Pod communications. You want a certain policy to be set at the Kubernetes cluster so that you can control how only certain Pods in certain namespaces are able to communicate with each other. How do you do it in Kubernetes? That is where you need to use network policy. A network policy defines the rules that control traffic on how the Pods should communicate with each other.

Like most other Kubernetes rules, network policies also rely on the label. Before we understand network policy with an example, you need to be aware of two terms ingress and egress. Ingress is used for incoming traffic, and egress is used for outgoing traffic. For ingress and egress, you can whitelist the sources of traffic like Pods, IP addresses, or ports.

The first step is to enable network policy in minikube by using Cilium. For this, first Delete your existing minikube cluster:

```
$ minikube delete
```

The above command will delete the minikube cluster.

Refer to Figure which shows the existing minikube is deleted:

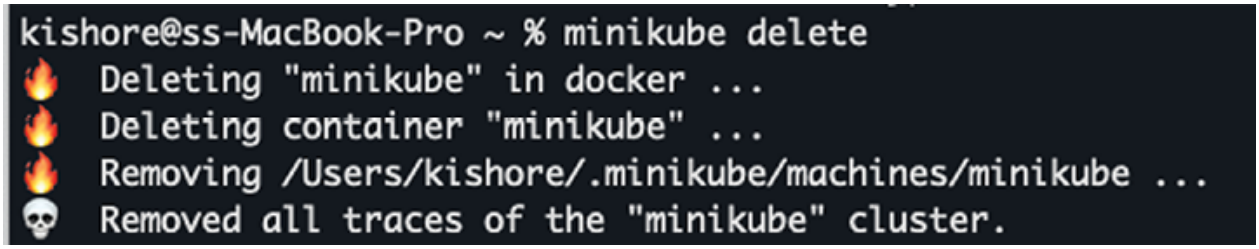

kishore@ss-MacBook-Pro ~ % minikube delete
Deleting "minikube" in docker ...
Deleting container "minikube" ...
Removing /Users/kishore/.minikube/machines/minikube ...
Removed all traces of the "minikube" cluster.

Figure 12.11: Minikube cluster is deleted

The next step is to start the minikube cluster with the network plugin option:

$minikube start --network-plugin=cni --enable-default-cni

The above command starts the minikube cluster with the network-plugin option.

Refer to Figure which shows the new minikube cluster created with the network plugin option:

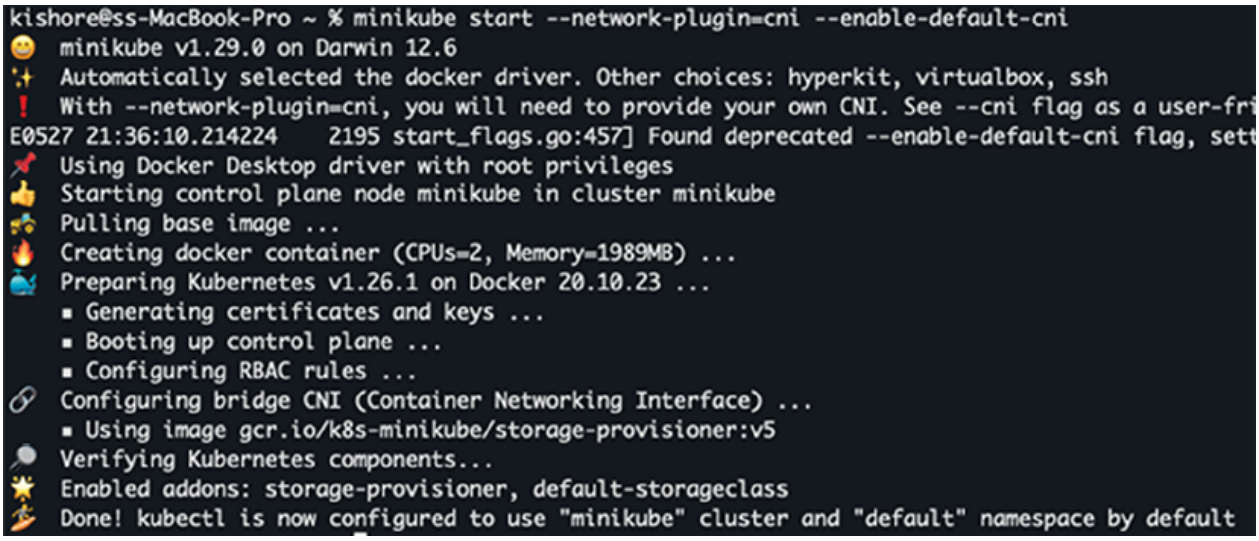

kishore@ss-MacBook-Pro ~ % minikube start --network-plugin=cni --enable-default-cni
minikube v1.29.0 on Darwin 12.6
Automatically selected the docker driver. Other choices: hyperkit, virtualbox, ssh
With --network-plugin=cni, you will need to provide your own CNI. See --cni flag as a user-fri
E0527 21:36:10.214224 2195 start_flags.go:457] Found deprecated --enable-default-cni flag, sett
Using Docker Desktop driver with root privileges
Starting control plane node minikube in cluster minikube
Pulling base image ...
Creating docker container (CPUs=2, Memory=1989MB) ...
Preparing Kubernetes v1.26.1 on Docker 20.10.23 ...
▪ Generating certificates and keys ...
▪ Booting up control plane ...
▪ Configuring RBAC rules ...
Configuring bridge CNI (Container Networking Interface) ...
▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
Verifying Kubernetes components...
Enabled addons: storage-provisioner, default-storageclass
Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default

Figure 12.12: Minikube cluster started with the network plugin option

The next step is to install cilium in the cluster

kubectl create -f

https://raw.githubusercontent.com/cilium/cilium/v1.9/install/kubernetes/quick-install.yaml

The above command will install cilium in the cluster.

Refer to Figure which shows the installation of cilium objects in the cluster:

Figure 12.13: Cilium installed in the cluster

If you check the Pods in the you will see the cilium Pods running, and the network policy is ready to use in minikube.

$kubectl get po -n kube-system

The above command shows Pods in the kube-system namespace.

Refer to Figure which shows the cilium Pods running in the kube-system namespace:

```
kishore@ss-MacBook-Pro ~ % kubectl get po -n kube-system
NAME                               READY   STATUS    RESTARTS         AGE
cilium-6kx29                       1/1     Running   0                101s
cilium-operator-7f67958d9f-h55p5   1/1     Running   0                101s
coredns-787d4945fb-fqcwk           1/1     Running   0                3m21s
etcd-minikube                      1/1     Running   0                3m34s
kube-apiserver-minikube            1/1     Running   0                3m34s
kube-controller-manager-minikube   1/1     Running   0                3m34s
kube-proxy-7blkn                   1/1     Running   0                3m22s
kube-scheduler-minikube            1/1     Running   0                3m34s
storage-provisioner                1/1     Running   1 (2m50s ago)    3m32s
```

Figure 12.14: Cilium Pods running in the kube-system namespace

Let us understand the network policy with examples. For our example, we will consider two Pods, Frontend, and Backend Pods. The Policy we need to define is only Frontend Pod can communicate with Backend Pod, whereas Backend Pod cannot talk to front end Pod.

Refer to Figure which shows the Policy which was explained above and how the traffic is flowing from the frontend Pod to the backend Pod:

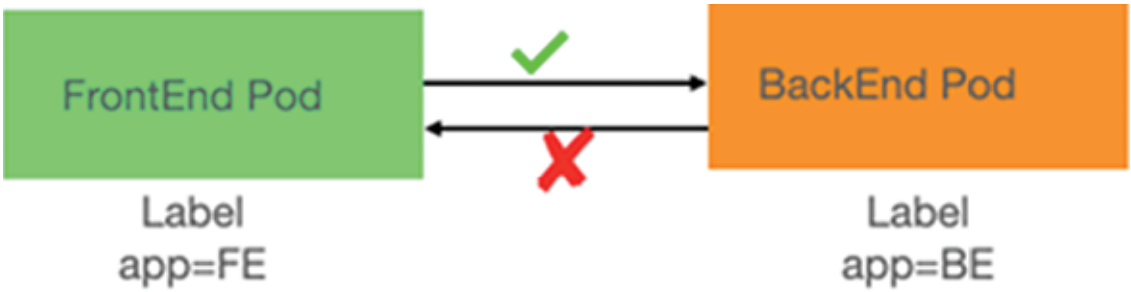


Figure 12.15: The flow of traffic from frontend to backend

The first step is to set up both the frontend and backend pods in the cluster. Let us first create the backend Pod:

$ kubectl run backend --image=nginx --restart=Never -l app=BE,role=api --port 80

The above command creates the backend Pod with the label

Refer to Figure which shows the Backend Pod is created imperatively:

```
kishore@ss-MacBook-Pro ~ % kubectl run backend --image=nginx --restart=Never \
  -l app=BE,role=api --port 80
pod/backend created
```

Figure 12.16: Backend Pod created imperatively

The following command shows the ip of the created

$kubectl get pods -o wide

Refer to Figure which shows the ip of the backend Pod:

```
kishore@ss-MacBook-Pro ~ % kubectl get pods -o wide
NAME      READY   STATUS    RESTARTS   AGE   IP           NODE
backend   1/1     Running   0          91s   10.0.0.165   minikube
```

Figure 12.17: ip of the backend Pod

The next step is to create a network policy declaratively. The following YAML manifest is for network Policy to be created:

apiVersion: networking.k8s.io/v1

```
kind: NetworkPolicy

metadata:

  name: api-flow

spec:

  podSelector:

    matchLabels:

      app: BE

  ingress:

  - from:

    - podSelector:

        matchLabels:

          app: FE
```

The network policy is created using create command.

```
$kubectl create -f networkpol.yaml
```

The above command creates a network policy. Refer to the following figure, which creates network policy:

```
kishore@ss-MacBook-Pro chapter12 % kubectl create -f networkpol.yaml
networkpolicy.networking.k8s.io/api-flow created
```

Figure 12.18: Network Policy created

The pod selector in YAML directs that the network policy is applicable to Pods with label which is back-end Pods in our case. Ingress to these back-end Pods is only the Pods whose labels match Any other Pod with labels other than an app:FE will not be able to communicate to the backend Pod.

The next step is to create a Pod with the label app=FE:

$kubectl run frontend --rm -it --image=busybox --restart=Never -l app=FE --/bin/sh

$wget --spider --timeout=1 10.0.0.165

The above command creates a Pod with the label app=FE, and once created, it can successfully connect to the backend Port via the wget command. Refer to Figure which shows successful connectivity:

```
kishore@ss-MacBook-Pro chapter12 % kubectl run  frontend --rm -it --image=busybox  --restart=Never -l app=FE -- /bin/sh
If you don't see a command prompt, try pressing enter.
/ #
/ #
/ # wget --spider --timeout=1 10.0.0.165
Connecting to 10.0.0.165 (10.0.0.165:80)
remote file exists
```

Figure 12.19: Front-end Pod with label FE connecting to back-end Pod

Next, we connect to create a Pod with the label app=API and try to connect to the backend Pod. As the label is not FE, network policy should kick in, and the Pod should not be able to connect to the backend Pod.

$kubectl run api --rm -it --image=busybox --restart=Never -l app=API --
/bin/sh

$ wget --spider --timeout=1 10.0.0.165

The above command creates a Pod with the label app=API, and once created, it is not able to connect to the backend Port via the wget command due to network policy. Refer to Figure which shows connectivity is not successful:

```
kishore@ss-MacBook-Pro chapter12 % kubectl run  api --rm -it --image=busybox  --restart=Never -l app=API -- /bin/sh
If you don't see a command prompt, try pressing enter.
/ #
/ #
/ # wget --spider --timeout=1 10.0.0.165
Connecting to 10.0.0.165 (10.0.0.165:80)
wget: download timed out
```

Figure 12.20: API Pod with label API not able to connect with backend Pod

Use the get command to list the network policy:

$kubectl get netpol

Refer to Figure which shows network policy in the cluster:

```
kishore@ss-MacBook-Pro chapter12 % kubectl get netpol
NAME       POD-SELECTOR   AGE
api-flow   app=BE         5m9s
```

Figure 12.21: Get network policy from the cluster

The following command describes the network policy:

$ kubectl describe netpol api-flow

Refer to Figure which describes the network policy:

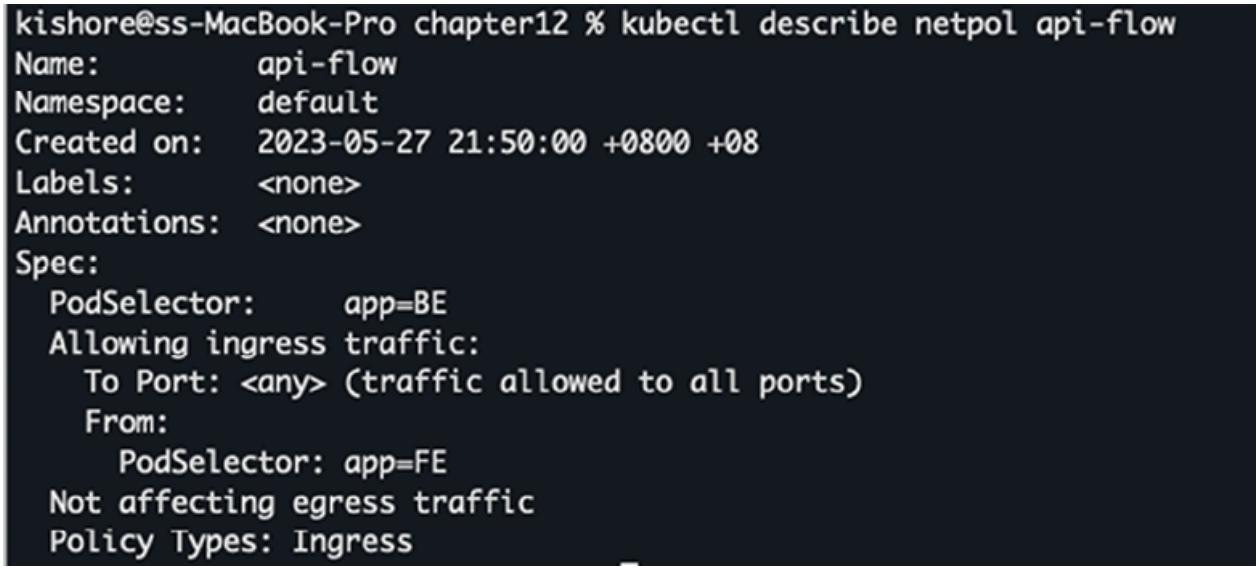


Figure 12.22: Describe network policy

If you want to disallow all the ingress and egress traffic, the following manifest YAML will do that. Note the curly braces for spec.podSelector means the policy is applicable to all Pods in the namespace.

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

```
  name: deny-all
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

If you want to restrict access on the port level, say you want to set a policy that Pod only exposes port 443 to the outside, the following manifest can be used to create a network policy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: port-access
spec:
  podSelector:
```

```
  matchLabels:
    app: BE
ingress:
- from:
  - podSelector:
      matchLabels:
        app: FE
  ports:
  - protocol: TCP
    port: 443
```

Troubleshooting Service

If you want to troubleshoot why Services are not getting mapped to the Pod, start by querying the Service and then render the labels of available Pods using command.

$kubectl describe service my-dep

The above command shows the service detail, including the selector options

Refer to Figure which shows the details of the Service. Note the selector has the value of

```
kishore@ss-MacBook-Pro ~ % kubectl describe svc my-dep
Name:              my-dep
Namespace:         default
Labels:            app=my-dep
Annotations:       <none>
Selector:          app=my-dep
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.100.147.111
IPs:               10.100.147.111
Port:              <unset>  80/TCP
TargetPort:        80/TCP
Endpoints:         10.0.0.115:80,10.0.0.243:80,10.0.0.30:80
Session Affinity:  None
Events:            <none>
```

Figure 12.23: Details of the Service

The following command shows the labels of the Pod:

$kubectl get pods --show-labels

Refer to Figure which shows the details of the Pods. Note the selector has a value of which is the same as the Service selector:

```
kishore@ss-MacBook-Pro ~ % kubectl get po  --show-labels
NAME                      READY   STATUS    RESTARTS   AGE     LABELS
my-dep-85974bb845-bhb82   1/1     Running   0          2m39s   app=my-dep
my-dep-85974bb845-dv2mj   1/1     Running   0          2m39s   app=my-dep
my-dep-85974bb845-mjwbq   1/1     Running   0          2m39s   app=my-dep
```

Figure 12.24: Details of the Pods with labels

You can also query the endpoints of the Service instance, which shows the ip of the Pods

$kubectl get endpoints my-dep

The above command will show the endpoint of the Service instance.

Refer to Figure which queries the Service endpoint Note the endpoint displays the IPs of the Pod:

```
kishore@ss-MacBook-Pro ~ % kubectl get endpoints my-dep
NAME     ENDPOINTS                                      AGE
my-dep   10.0.0.115:80,10.0.0.243:80,10.0.0.30:80   4m56s
```

Figure 12.25: Display endpoint

Conclusion

In this chapter, we looked at various types of Kubernetes Services and learned about Kubernetes networking and network policy. The next two chapters are focused on the CKAD mock exams.

Questions

Create a deployment named The deployment should have two replicas, and the image should be based on nginx. Create a Service based on NodePort. The Service port should be 8080. The website should be accessible from port 32001 to all hosts.

For this question, assume a two-tier application that has an app layer and a database layer. Both layers are deployed in the Pod BE app and data app. Below is the YAML manifest of the Pod definition. Create the two Pods and define a network policy so that only the app Pod can communicate with the data Pod. All these should be created under the namespace

kind: Pod

apiVersion: v1

metadata:

name: backend

namespace: application

labels:

app: be

```
spec:
  containers:

  - name: backend
    image: nginx
kind: Pod
apiVersion: v1
metadata:
  name: database
  namespace: application
  labels:
    app: data
spec:
  containers:
```

```
- name: database

  image: mysql

  env:

  - name: MYSQL_ROOT_PASSWORD

    value: password
```

Solutions

Solution to question 1:

For the solution, create the deployment test Service imperatively:

$kubectl create deployment test-service --image=nginx --replicas=2

The next step is to create a declaratively using manifest YAML:

```
apiVersion: v1

kind: Service

metadata:

   name: myservice

   labels:

     run: myservice

 spec:

     type: NodePort
```

```
  ports:
  -  port: 8080
     targetPort: 80


     nodePort: 32001
     protocol: TCP
     selector:
        run: test-service
```

Solution to question 2:

First, create a namespace application:

```
$kubectl create ns application
```

Create the Pod definition using manifest YAML provided in the question.

Create a network policy declarative using the Manifest YAML, and specify a rule that only the backend app can communicate to the db app:

```
apiVersion: networking.k8s.io/v1
```

```
kind: NetworkPolicy
metadata:
  name: app-stack-network-policy
  namespace: application
spec:
  podSelector:

    matchLabels:
      app: data
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
```

```
  - podSelector:

      matchLabels:

        app: be
```

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 13

Mock Exam 1

Introduction

It is time for sample exam 1. The questions are intended to be completed within two hours. If you cannot do it within two hours, you must increase your speed before taking the exam. All the best.

Structure

In this chapter, we will cover the following topics:

Mock exam

Objectives

The objective of this chapter is to expose you to sample CKAD exam questions, which need to be completed within the stipulated time of two hours. Let us get started.

Questions

Question 1: Create a namespace called In this namespace, host a Pod with image nginx called

Create a namespace called In this namespace, create a busy box Pod called pod2, which runs the command

Get Pods on all namespaces.

Answer: The first step is to create namespace ckadpractice1 using the kubectl create command:

$kubectl create ns ckadpractice1

The above command creates namespace The next step is to create a Pod using the kubectl command in the newly created namespace

$kubectl run pod1 --image=nginx --namespace=ckadpractice1

The above command creates pod1 in the namespace

Refer to Figure which shows how namespace and Pod is created:

```
kishore@ss-MacBook-Pro ~ % kubectl create ns ckadpractice1
namespace/ckadpractice1 created
kishore@ss-MacBook-Pro ~ % kubectl run pod1 --image=nginx  --namespace=ckadpractice1
pod/pod1 created
```

Figure 13.1: Namespace and Pod created

Similar to namespace create namespace ckadpractice2 using kubectl command:

$kubectl run pod2 -–image=busybox -–restart=Never -–namespace=ckadpractice2 --command --env

The above command creates a Pod in namespace

Refer to Figure which shows how namespace and Pod are created in namespace

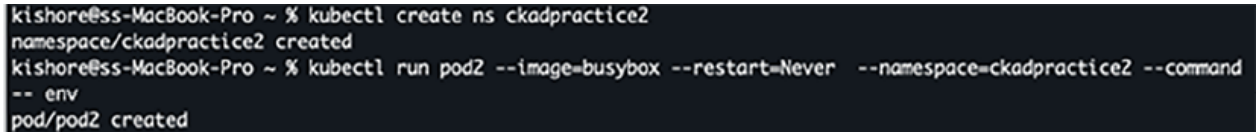

kishore@ss-MacBook-Pro ~ % kubectl create ns ckadpractice2
namespace/ckadpractice2 created
kishore@ss-MacBook-Pro ~ % kubectl run pod2 --image=busybox --restart=Never --namespace=ckadpractice2 --command
-- env
pod/pod2 created

Figure 13.2: Namespace and Pod created

Refer to Figure which shows the log output for the busybox

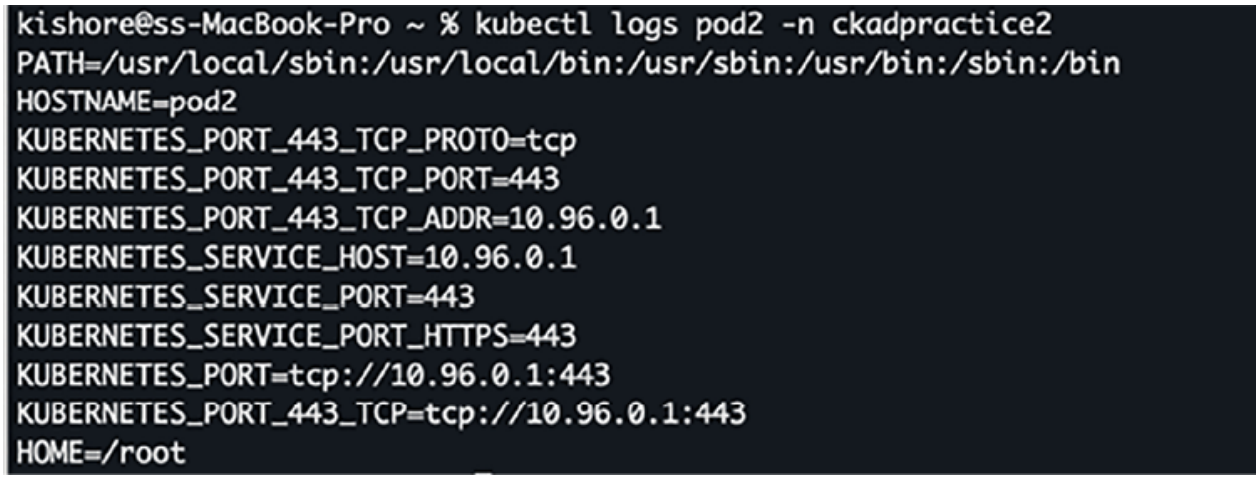


Figure 13.3: Log output of the Pod

Next is to get Pods on all the

$kubectl get po --all-namespaces

The above command displays Pod across all namespaces.

Refer to Figure which shows a display of Pods across all namespaces:

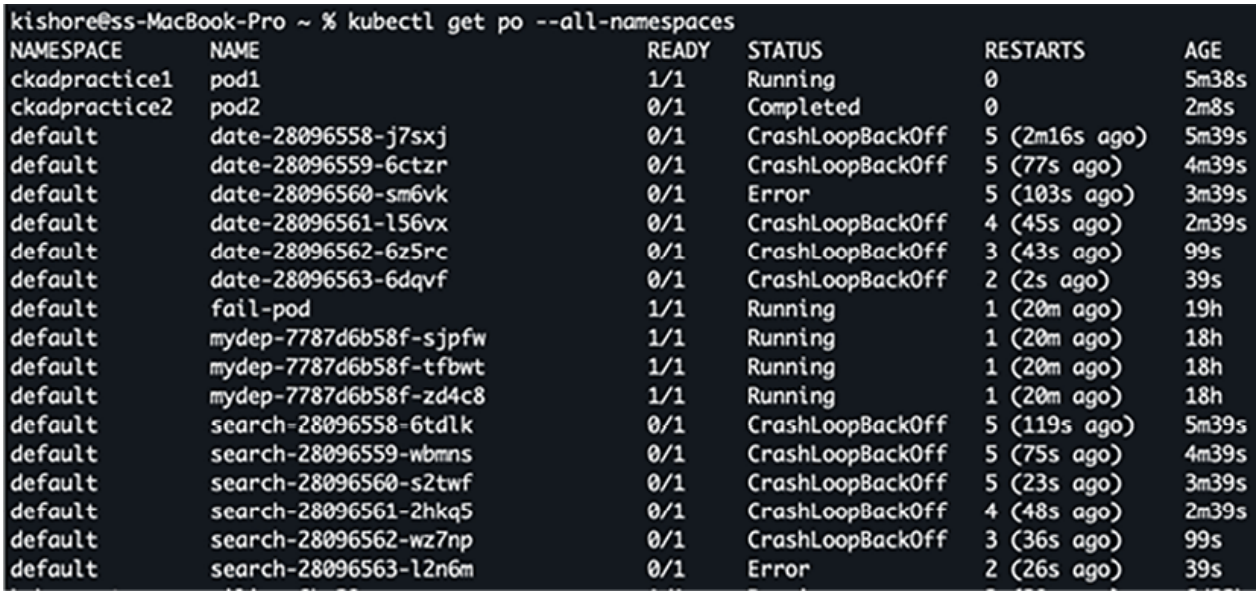

kishore@ss-MacBook-Pro ~ % kubectl get po --all-namespaces
NAMESPACE NAME READY STATUS RESTARTS AGE
ckadpractice1 pod1 1/1 Running 0 5m38s
ckadpractice2 pod2 0/1 Completed 0 2m8s
default date-28096558-j7sxj 0/1 CrashLoopBackOff 5 (2m16s ago) 5m39s
default date-28096559-6ctzr 0/1 CrashLoopBackOff 5 (77s ago) 4m39s
default date-28096560-sm6vk 0/1 Error 5 (103s ago) 3m39s
default date-28096561-l56vx 0/1 CrashLoopBackOff 4 (45s ago) 2m39s
default date-28096562-6z5rc 0/1 CrashLoopBackOff 3 (43s ago) 99s
default date-28096563-6dqvf 0/1 CrashLoopBackOff 2 (2s ago) 39s
default fail-pod 1/1 Running 1 (20m ago) 19h
default mydep-7787d6b58f-sjpfw 1/1 Running 1 (20m ago) 18h
default mydep-7787d6b58f-tfbwt 1/1 Running 1 (20m ago) 18h
default mydep-7787d6b58f-zd4c8 1/1 Running 1 (20m ago) 18h
default search-28096558-6tdlk 0/1 CrashLoopBackOff 5 (119s ago) 5m39s
default search-28096559-wbmns 0/1 CrashLoopBackOff 5 (75s ago) 4m39s
default search-28096560-s2twf 0/1 CrashLoopBackOff 5 (23s ago) 3m39s
default search-28096561-2hkq5 0/1 CrashLoopBackOff 4 (48s ago) 2m39s
default search-28096562-wz7np 0/1 CrashLoopBackOff 3 (36s ago) 99s
default search-28096563-l2n6m 0/1 Error 2 (26s ago) 39s

Figure 13.4: Pod across all namespace

Question 2: Create a Pod with the name testpod with image nginx and expose traffic on port 80.

Update the Pod image to

Get IP of the Pod testpod, which was created in earlier step.

Answer: The first step is to create Pod with image nginx and expose traffic on port 80:

$kubectl run testpod --image=nginx --port=80

The above command creates Pod with nginx image and exposes traffic on port 80.

Refer to Figure which creates the Pod and exposes the traffic on port 80:

```
kishore@ss-MacBook-Pro ~ % kubectl run testpod --image=nginx --port=80
pod/testpod created
kishore@ss-MacBook-Pro ~ % kubectl get po testpod
NAME      READY   STATUS    RESTARTS   AGE
testpod   1/1     Running   0          17s
```

Figure 13.5: Pod created, and port exposed for traffic

The next step is to update the image:

$ kubectl set image pod testpod testpod=nginx:1.7.1

The above command updates the image of nginx to 1.7.1

Refer to Figure which shows the image of the container being updated:

kishore@ss-MacBook-Pro ~ % kubectl set image pod testpod testpod=nginx:1.7.1
pod/testpod image updated
kishore@ss-MacBook-Pro ~ % kubectl describe po testpod | grep -i image
Image: nginx:1.7.1
Image ID: docker-pullable://nginx@sha256:af296b188c7b7df99ba960ca614439c99cb7cf252ed7bbc23e90cfda590923
05
Normal Pulling 4m13s kubelet Pulling image "nginx"
Normal Pulled 4m11s kubelet Successfully pulled image "nginx" in 2.41837466s (2.945544743s inc
luding waiting)

Figure 13.6: Update image of the Pod

The following command gets the IP details of the Pod:

$ kubectl get po testpod -o wide

Refer to Figure which shows the IP of the Pod:

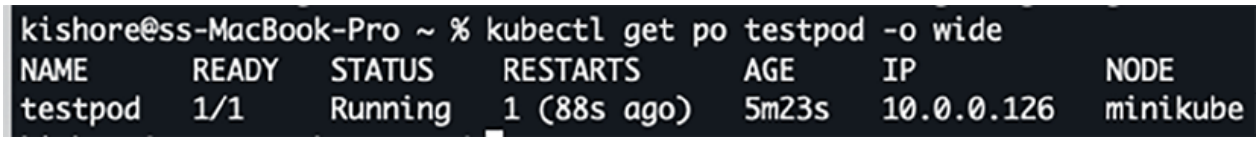

kishore@ss-MacBook-Pro ~ % kubectl get po testpod -o wide
NAME READY STATUS RESTARTS AGE IP NODE
testpod 1/1 Running 1 (88s ago) 5m23s 10.0.0.126 minikube

Figure 13.7: IP address of Pod

Question 3: Continuing with the previous question, get the logs of the Pod

Next, get logs about the previous instance of Pod

Execute a simple shell on the

Answer: The first step is to get the log of the Pod:

$ kubectl logs testpod

The above command outputs the logs of the Pod

$ kubectl logs testpod -p

The above command, with option outputs the logs of the previous instance of the Pod.

Refer to Figure which shows the output log of the Pod:

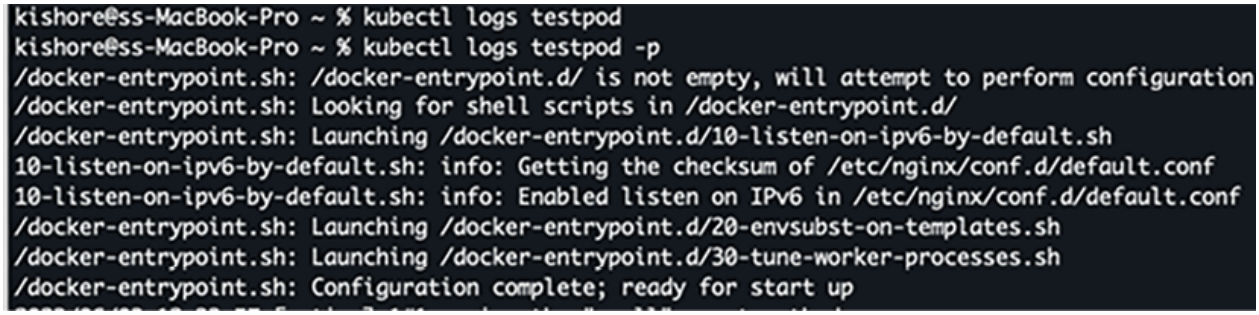

kishore@ss-MacBook-Pro ~ % kubectl logs testpod
kishore@ss-MacBook-Pro ~ % kubectl logs testpod -p
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up

Figure 13.8: Log output of the Pod

The following command executes a simple shell on the Pod:

$ kubectl exec -it testpod -- /bin/sh

Refer to Figure which shows execute the command on the Pod:

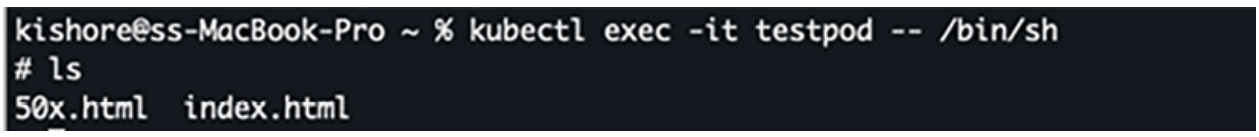


Figure 13.9: Execute command on the Pod

Question 4: Create a busybox image Pod with the name temppod that echoes and then delete the Pod automatically once the Pod is in a complete state.

Answer: The following command creates a busybox image Pod with the name temppod and deletes it automatically once completed:

kubectl run temppod –-image=busybox -it —rm -–restart=Never -- /bin/sh -c ‘echo hello’

Question 5: Create a Pod envpod with image nginx, and in the Pod, set an environment value as

Answer: The following kubectl command creates Pod and sets environment variable user with value admin:

$kubectl run envpod --image=nginx --env=user=admin

Refer to Figure which creates the Pod and sets the environment variable:

```
kishore@ss-MacBook-Pro ~ % kubectl run envpod --image=nginx --env=user=admin
pod/envpod created
kishore@ss-MacBook-Pro ~ % kubectl exec -it envpod -- env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=envpod
TERM=xterm
user=admin
MYDEP_SERVICE_HOST=10.97.113.142
```

Figure 13.10: Pod set with environment variable

Question 6: Create a multi-container Pod, the Pod should contain two containers, both with image busybox, and the Pod should execute the

command echo hello; sleep

Answer: The best way to create a multi-Pod is to create it declarative from YAML:

```
apiVersion: v1

kind: Pod

metadata:

  name: multipod

spec:

  containers:

  - args:

    - /bin/sh

    - -c

    - echo hello;sleep 2400


    image: busybox

    name: busybox1
```

```
- args:
  - /bin/sh
  - -c
  - echo hello;sleep 2400
  image: busybox
  name: busybox2
```

Next step, use kubectl create command to create the multi container Pod.

Question 7: Create three pods with names pod1, pod2, and pod3, all with same image nginx. All of them should have the label release=v1

Show all labels of the Pods.

Change the labels of pod2 to be release=v2

Answer: The first step is to create three pods with labels mentioned in the question:

```
$kubectl run pod1 --image=nginx --labels=release=v1
```

The above command creates a Pod pod1 with label Execute the similar command for pod2 and

$kubectl get po --show-labels

The above command shows Pod details with the label.

Refer to Figure which shows how the Pods are created with the label:

```
kishore@ss-MacBook-Pro ~ % kubectl run pod1 --image=nginx --labels=release=v1
pod/pod1 created
kishore@ss-MacBook-Pro ~ % kubectl run pod2 --image=nginx --labels=release=v1
pod/pod2 created
kishore@ss-MacBook-Pro ~ % kubectl run pod3 --image=nginx --labels=release=v1
pod/pod3 created
kishore@ss-MacBook-Pro ~ % kubectl get po --show-labels
NAME   READY   STATUS    RESTARTS   AGE   LABELS
pod1   1/1     Running   0          44s   release=v1
pod2   1/1     Running   0          38s   release=v1
pod3   1/1     Running   0          29s   release=v1
```

Figure 13.11: Pod created with label

The following command updates the value of the label release to v2 for

$kubectl label po pod2 release=v2 --overwrite

Refer to Figure which updates the value of the label of the Pod:

```
kishore@ss-MacBook-Pro ~ % kubectl label po pod2 release=v2 --overwrite
pod/pod2 labeled
kishore@ss-MacBook-Pro ~ % kubectl get po --show-labels
NAME   READY   STATUS    RESTARTS   AGE   LABELS
pod1   1/1     Running   0          96s   release=v1
pod2   1/1     Running   0          90s   release=v2
pod3   1/1     Running   0          81s   release=v1
```

Figure 13.12: Update to Pod label

Question 8: Continuing with question 7, for the created Pods get only the Pods

Remove the label from the Pods we created before as part of question 7.

Answer: The following command gets only Pods with label

$kubectl get po -l release=v2

The above command removes the label from all the Pods:

$kubectl label po pod1 pod2 pod3 release-

Question 9: Continuing with question 8, annotate pod1, pod2, pod3 with

Remove the annotations from the three Pods.

Remove the Pods from the cluster.

Answer: The following command annotates the pod1, pod2 and pod3:

$kubectl annotate po pod1 pod2 pod3 author= ‘myname’

The following command removes the annotates from pod and

$kubectl annotate po pod1 pod2 pod3 author-

$kubectl delete po pod1 pod2 pod3

The above deletes the Pods pod2 and pod3 from the cluster:

Question 10: Create a deployment mydep with image nginx, having 2 replicas.

Update the nginx image of the mydep to

Scale the deployment mydep from existing 2 to 4 replicas.

Answer: The first step is to create the deployment with image nginx with two replicas:

$kubectl create deployment mydep --image=nginx --replicas=2

The above command creates deployment with 2 replicas.

Refer to Figure which shows the creation of deployment

```
kishore@ss-MacBook-Pro ~ %  kubectl create deployment mydep --image=nginx --replicas=2
deployment.apps/mydep created
kishore@ss-MacBook-Pro ~ % kubectl get deployment
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
mydep   2/2     2            2           10s
```

Figure 13.13: Creation of deployment object

The next step is to update the

$kubectl set image deploy mydep nginx=nginx:1.19.8

The above command updates the nginx

Refer to Figure which shows updating of the image:

```
kishore@ss-MacBook-Pro ~ % kubectl set image deploy  mydep  nginx=nginx:1.19.8
deployment.apps/mydep image updated
kishore@ss-MacBook-Pro ~ % kubectl describe deploy mydep | grep -i image
    Image:        nginx:1.19.8
```

Figure 13.14: Update the image

The next step is to scale the deployment:

$kubectl scale deploy mydep --replicas=4

The above command scales the deployment to 4 replicas.

Question 11: Create a job named my-image with image busybox that runs the command

Answer: The following command creates the job which outputs the commands date:

$kubectl create job my-job --image=busybox -- date

Question 12: Create a cron job my-cron with image busybox that runs on a schedule of * * * and writes ‘ echo Hello from the K8S ‘.

Answer: The following commands create the which is executed on schedule and executes the echo command:

$kubectl create cronjob my-cron --image=busybox --schedule=”*/1 * * * *” -- /bin/sh -c ‘echo Hello from the K8S’

Question 13: Create a basic Helm chart chart1 and Find pending Helm deployments on all namespaces.

Answer: The first step is to create a helm chart:

$helm create chart1

The above command creates the helm

$helm list --pending -A

The above command lists down the pending helm deployments across all the namespaces.

Question 14: Create a ConfigMap called with the value Create a new nginx Pod with name configpod that loads the value from variable in an env variable called

Answer: The first step is to create configMap configtest using the kubectl command:

$kubectl create cm configtest --from-literal=var=value1

The above command creates the ConfigMap from literal values.

Refer to Figure which shows how ConfigMap is created using the kubectl command:

```
kishore@ss-MacBook-Pro ~ % kubectl create cm configtest --from-literal=var=value1
configmap/configtest created
kishore@ss-MacBook-Pro ~ % kubectl get cm
NAME                DATA   AGE
configtest          1      6m12s
```

Figure 13.15: ConfigMap created

The next step is using the config map in the Pod that loads the value from variable in an env variable called Create the Pod using the below YAML:

apiVersion: v1

kind: Pod

```
metadata:
  labels:
  name: configpod
spec:
  containers:
  - image: nginx
    name: nginx
    env:
    - name: configenv # name of the env variable

      valueFrom:
        configMapKeyRef:
          name: configtest # name of config map
          key: var # name of the entity in config map
```

The next step is to use kubectl create command to create the Pod.

Question 15: Create a secret called secret-test with the values password=passw0rd1

Answer: The above command creates a secret from

$ kubectl create secret generic secret-test --from-literal=password=passw0rd1

Question 16: Create a new service account test1 and use this service account in Pod nginx.

Answer: The first step is to create service account using kubectl command:

$kubectl create sa test1

The above command creates service account

Once the service account is created the next step is to create the Pod using YAML and specifying the created service account test1 in the YAML definition:

apiVersion: v1

kind: Pod

metadata:

  labels:

```
  name: nginx

spec:

  serviceAccountName: test1

containers:

  - image: nginx

    name: nginx
```

Next step, use kubectl create command to create the Pod.

Question 17: Create a namespace rt-test and create Resource quota in this namespace one with hard requests memory=2Gi and hard limits

Answer: The first step is to create the namespace

$kubectl create ns rt-test

The above command creates a namespace using kubectl command.

The next step is to create a resource quota using YAML with request and limits as mentioned in the question:

```
apiVersion: v1
```

```
kind: ResourceQuota

metadata:

  name: test-rq

  namespace: rt-test

spec:

  hard:

    requests.cpu: 2

    requests.memory: 2Gi

    limits.cpu: 4

    limits.memory: 4Gi
```

Next step, use kubectl create command to create the Resource quota.

Question 18: Create a Pod with name livetest with image nginx, with a liveness probe that runs the command

Answer: The Pod with liveness probe can be created using the YAML:

```
apiVersion: v1
```

```
kind: Pod

metadata:

  name: livetest

spec:

  containers:

  - image: nginx

    name: livetest

    livenessProbe:

    exec:

      command:

      - ls
```

Next step, use kubectl create command to create the Pod.

Question 19: Create a PersistentVolume of called Make it have accessMode of storageClassName mounted on hostPath

Answer: Persistent volume can be created by YAML.

```
kind: PersistentVolume

apiVersion: v1

metadata:

  name: mypv

spec:

  storageClassName: normal

  capacity:

    storage: 5Gi

  accessModes:

    - ReadWriteOnce

  hostPath:

    path: /etc/foo
```

Next step, use kubectl create command to create the persistent volume.

Conclusion

In this chapter, we focused on CKAD mock questions. In next chapter, we will continue with CKAD mock exam.

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 14

Mock Exam 2

Introduction

It is time for a sample exam 2. The questions are intended to be completed within two hours. If you cannot do it within two hours, you must increase your speed before taking the exam. All the best.

Structure

In this chapter, we will cover the following topics:

Mock exam

Objectives

The objective of this chapter is to expose you to sample CKAD exam questions, which need to be completed within the stipulated time of two hours.

Questions

Question 1: Create the namespace

In the namespace create a new Pod named testpod with the latest image of The Pod should be exposed at port 80.

Is the container being created correctly? If not, add your observation in the text file.

The image of the pod needs to be updated to the 1.15.12 version of nginx.

Answer: The first step is to create namespace

$kubectl create ns ckad-test

The above command creates a namespace

Refer to Figure which creates a namespace

```
kishore@ss-MacBook-Pro ~ % kubectl create ns ckad-test
namespace/ckad-test created
```

Figure 14.1: Namespace created

The next step is to create pod testpod in the namespace

$kubectl run testpod --image=nginx:1.99 --restart=Never --port=80 --namespace=ckad-test

The above command creates a pod testpod and exposes port

$kubectl get pods -n ckad-test

The above command gets the status of the created

Refer to Figure which creates the Pod and checks its status:

```
kishore@ss-MacBook-Pro ~ % kubectl run testpod --image=nginx:1.99 --restart=Never --port=80 --namespace=ckad-test
pod/testpod created
kishore@ss-MacBook-Pro ~ % kubectl get pods -n ckad-test
NAME      READY   STATUS         RESTARTS   AGE
testpod   0/1     ErrImagePull   0          14s
```

Figure 14.2: Create Pod and check its status

As the status of the Pod shows that is, the image is not found.

The next step is to write this error message in the error.txt file:

$ echo “ImagePullBackOff” | cat >>error.txt

Refer to Figure which writes the error in the error.txt file:

```
kishore@ss-MacBook-Pro ~ % echo "ImagePullBackOff" | cat >>error.txt
kishore@ss-MacBook-Pro ~ % cat error.txt
ImagePullBackOff
```

Figure 14.3: Document error

To change the image of the Pod, use the kubectl edit command,

$kubectl edit pod testpod -n ckad-test

The above command edits the live Pod testpod.

Refer to Figure in which the image of the Pod is edited to value 1.15.12:

```
# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving this file will be
# reopened with the relevant failures.
#
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: "2023-06-02T12:52:16Z"
  labels:
    run: testpod
  name: testpod
  namespace: ckad-test
  resourceVersion: "75473"
  uid: 78fd3c9f-6eab-49e5-9bea-5a71160d7eca
spec:
  containers:
  - image: nginx:1.15.12
    imagePullPolicy: IfNotPresent
    name: testpod
    ports:
    - containerPort: 80
      protocol: TCP
    resources: {}
    terminationMessagePath: /dev/termination-log
```

Figure 14.4: Edit running Pod

Refer to the following figure, which shows the pod test-pod status running:

```
kishore@ss-MacBook-Pro ~ % kubectl edit pod testpod -n ckad-test
pod/testpod edited
kishore@ss-MacBook-Pro ~ % kubectl get pods -n ckad-test
NAME      READY   STATUS    RESTARTS   AGE
testpod   1/1     Running   0          3m11s
```

Figure 14.5: Status of the Pod

Question 2: Continuing with question 1, for the testpod created, do sh into the container and run the command to list files/directories. Note down the output of the command in the out.txt file.

Retrieve the IP address of the Pod testpod.

Use the image busybox and run a temporary Pod. Once the Pod runs, shell into the container and run a wget command to query testpod created in question 1.

Delete the Pod testpod and the namespace

Answer: Log into the testpod and execute the ls command:

$kubectl exec testpod -it --namespace=ckad-test -- /bin/sh

The above command shells into the testpod.

Refer to Figure which executes the testpod and executes the ls command.

```
kishore@ss-MacBook-Pro ~ % kubectl exec testpod -it --namespace=ckad-test  -- /bin/sh
# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
# exit
```

Figure 14.6: exec pod and list dirs

The next step is to write the output of the ls command from the previous step to the out.txt file:

$vi out.txt

The above command creates the out.txt file where you can add the output of the ls command.

Figure 14.7 lists the output of the ls command:

Figure 14.7: Add the list of dirs to out.txt

The next step is to get the ip of the Pod testpod:

$kubectl get pods -o wide -n ckad-test

The above command lists the ip of the Pod.

Figure 14.8 list down the ip of pod

```
kishore@ss-MacBook-Pro ~ % kubectl get pods -o wide -n ckad-test
NAME      READY   STATUS    RESTARTS   AGE   IP           NODE
testpod   1/1     Running   0          46m   10.0.0.183   minikube
```

Figure 14.8: List the Pod IP

The next step is to create a temporary Pod with busybox image and query the testpod:

$kubectl run busybox --image=busybox --rm -it --restart=Never -n ckad-test -- /bin/sh

The above command creates the busybox pod, --rm is used to create this as a temporary Pod:

$wget -O- 10.0.0.183

The above command runs the wget command using the testPod IP.

Figure 14.9 shows the creation of the temporary Pod of busybox, shell into the temporary Pod, and run a wget command to query the testpod:

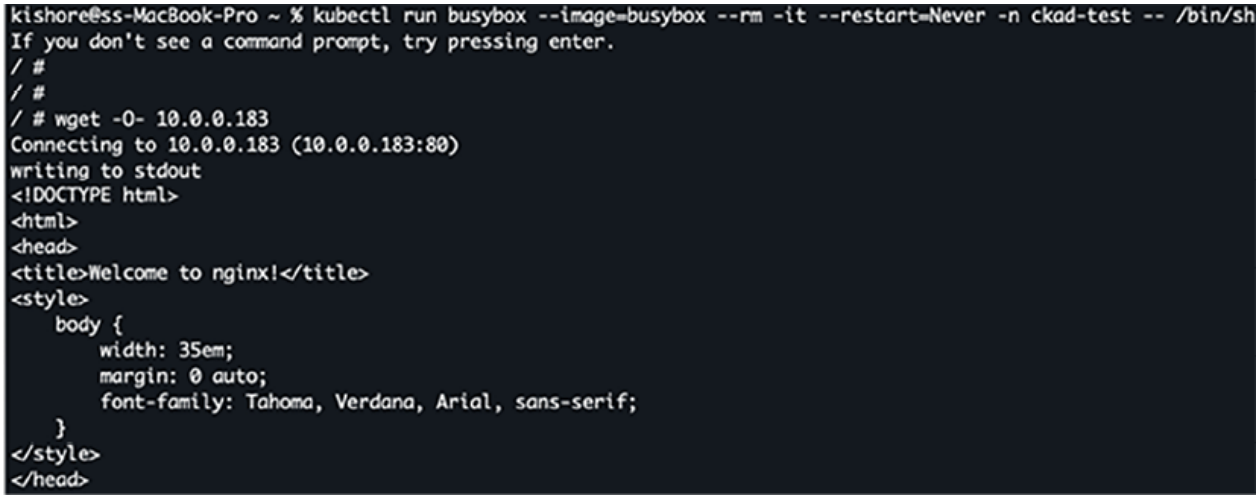


Figure 14.9: Run busybox pod to query testpod

The next step is to delete Pod and

$kubectl delete pod testpod --namespace=ckad-test

$kubectl delete ns ckad-test

The above two commands delete the Pod and

Refer to Figure which shows the Pod and namespaces deleted:

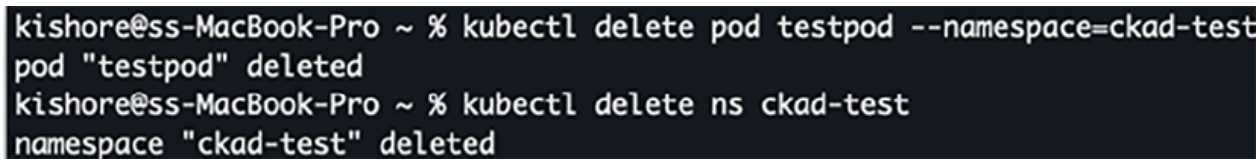


Figure 14.10: Pod and Namespace deleted

Question 3: Create a new file named The file will have the below-mentioned environment variables in a key/value pair format.

DB_PORT equates to 6603.

DB_USER equates to admin.

Create a new ConfigMap named dbtest-config from the configtest.txt file.

Pod named db to be created with image nginx that uses the environment variables from the

You need to verify that newly created environment variables are present in the Pod.

Answer: First, create a file named configtest with environment variables mentioned in the question:

$echo -e “DB_PORT=6603\nDB_USER=admin” > configtest.txt

The above command creates a file configtest with environment variables.

Figure 14.11 shows how the file configtest is created:

Figure 14.11: File created

The next step is to create the config map from the environment file imperatively:

$kubectl create configmap dbtest-config --from-env-file=configtest.txt

The above command creates the configMap

Figure 14.12 creates a configMap dbtest-config from the file

kishore@ss-MacBook-Pro ~ % kubectl create configmap dbtest-config --from-env-file=configtest.txt
configmap/dbtest-config created

Figure 14.12: ConfigMap created

Next, create the Pod using the YAML file. This Pod uses the environment variables from configMap

apiVersion: v1

kind: Pod

metadata:

  name: db

spec:

  containers:

```
  - image: nginx

    name: db


    envFrom:

      - configMapRef:

          name: dbtest-config
```

Next, create the Pod declaratively from the YAML using kubectl command.

The next step is to shell into the Pod db and print out the environment variables:

```
$kubectl exec db –-env
```

The above command shells to the db Pod and prints out the environment variable.

Figure 14.13 shows the print output of the environment variable:

```
kishore@ss-MacBook-Pro chapter14 % kubectl get pod db
NAME   READY   STATUS    RESTARTS   AGE
db     1/1     Running   0          53s
kishore@ss-MacBook-Pro chapter14 % kubectl exec db -- env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=db
DB_PORT=6603
DB_USER=admin
MY_DEP_SERVICE_PORT=80
MY_DEP_PORT=tcp://10.100.147.111:80
MY_DEP_SERVICE_HOST=10.100.147.111
```

Figure 14.13: Output of environment variables

Question 4: The Kubernetes secret object named db-secret must be created with the key/value pair

Create a nginx Pod named dbsecretpod that defines the secret db-secret as an environment variable. The environment variable will be named

Answer: The first step is to create the secret imperatively.

$kubectl create secret generic db-secret --from-literal=password=passw0d

The above command creates secret db-secret imperatively.

Figure 14.14 shows the secret is created:

```
kishore@ss-MacBook-Pro chapter14 % kubectl create secret generic db-secret --from-literal=password=passw0d
secret/db-secret created
kishore@ss-MacBook-Pro chapter14 % kubectl get secrets
NAME        TYPE     DATA   AGE
db-secret   Opaque   1      54s
```

Figure 14.14: Created secret db

The next step is to create Pod using YAML and kubectl create command. The YAML is defined by the secret to be used:

```
apiVersion: v1

kind: Pod

metadata:

  name: dbsecretpod

spec:

  containers:



  - image: nginx

    name: dbsecretpod

    env:

      - name: DB_PASS

        valueFrom:

          secretKeyRef:
```

name: db-secret

key: password

Figure 14.15 shows the Pod is created and secretly used as an environment variable:

```
kishore@ss-MacBook-Pro chapter14 % kubectl get pod dbsecretpod
NAME          READY   STATUS    RESTARTS   AGE
dbsecretpod   1/1     Running   0          51s
kishore@ss-MacBook-Pro chapter14 % kubectl exec dbsecretpod -- env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=dbsecretpod
DB_PASS=passw0d
```

Figure 14.15: Pod created with a secret as an environment variable

Question 5: Create a Pod named securedpod that uses the image nginx. an emptyDir volume to be mounted onto the directory /data/testapp

The filesystem group ID 4000 be set for the files created on this volume.

Answer: The best way to answer this question is by creating the Pod declaratively, create a Pod using YAML, which has the security context defined:

apiVersion: v1

kind: Pod

```
metadata:
  name: securedpod
spec:
  securityContext:
    fsGroup: 4000
  containers:
  - image: nginx
    name: securedcontainer
    volumeMounts:
    - name: data-vol
      mountPath: /data/testapp
  volumes:
  - name: data-vol
    emptyDir: {}
```

Next, use the kubectl create command to create the Pod.

Question 6: Resource quota-named apps are to be created under the namespace demo using the below YAML definition.

```
apiVersion: v1

kind: ResourceQuota

metadata:

  name: app

spec:

  hard:

    pods: “1”

    requests.cpu: “1”

    requests.memory: 500m
```

Run a new Pod which exceeds the limits of the above-mentioned resource quota requirements.

Answer: The first step is to create a namespace demo and resource quota with YAML provided.

Figure 14.16 shows how the namespace demo is created, and a corresponding resource quota object is created:

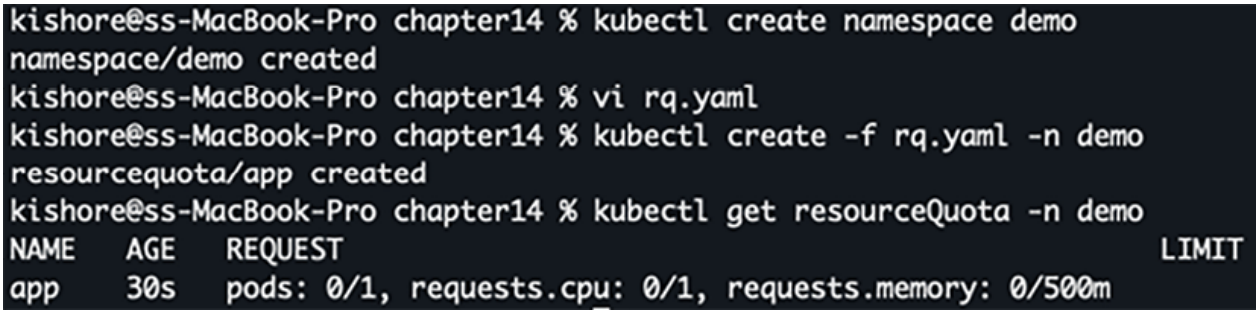

kishore@ss-MacBook-Pro chapter14 % kubectl create namespace demo
namespace/demo created
kishore@ss-MacBook-Pro chapter14 % vi rq.yaml
kishore@ss-MacBook-Pro chapter14 % kubectl create -f rq.yaml -n demo
resourcequota/app created
kishore@ss-MacBook-Pro chapter14 % kubectl get resourceQuota -n demo
NAME AGE REQUEST LIMIT
app 30s pods: 0/1, requests.cpu: 0/1, requests.memory: 0/500m

Figure 14.16: Namespace and resource quota created

Next, create a pod using YAML which exceeds the resource quota limit:

```
apiVersion: v1

kind: Pod

metadata:

  name: pod1

spec:

  containers:

  - image: nginx

    name: cont1
```

```
    resources:
      requests:
        memory: "1G"
        cpu: "400m"
```

Use the kubectl create command to create the Pod. Refer to Figure 14.17 shows that the Pod is not able to create as it exceeds the resource quota:

kishore@ss-MacBook-Pro chapter14 % kubectl create -f pod4.yaml -n demo
Error from server (Forbidden): error when creating "pod4.yaml": pods "pod1" is forbidden: exceeded quota: app,
memory=1G, used: requests.memory=0, limited: requests.memory=500m

Figure 14.17: Pod failed to create as it exceeds the resource quota

Question 7: A service account named sa needs to be created.

Pod named podsa uses the service account The Pod should create with image nginx.

Answer: The following kubectl create command creates a service account:

$kubectl create serviceaccount sa

The next step is to create Pod declaratively using YAML and adding service account sa to the Pod definition:

```
apiVersion: v1

kind: Pod

metadata:

  name: podsa

spec:

  serviceAccountName: sa

  containers:



  - image: nginx

    name: cont1
```

Use the kubectl create command to create the Pod from the YAML.

Question 8: Create a new multi-container Pod in a YAML.

The first container appln uses the image busybox and runs the below command.

'while true; do echo “$(date) | $(du -sh ~)” >> /var/logs/space.txt; sleep 10; done;

The second adapter container transform uses the image busybox and runs the command.

sleep 30; while true; do while read LINE; do echo “$LINE” | cut -f2 -d”|” >> $(date +%Y-%m-%d-%H-%M-%S)-transform.txt; done < /var/logs/space.txt; sleep 20; done;

Answer: Create the multi-container Pod from YAML definition:

```
apiVersion: v1

kind: Pod

metadata:

  name: adapter

spec:



  volumes:

    - name: config-volume

      emptyDir: {}

  containers:
```

```
- args:
  - /bin/sh
  - -c
  - 'while true; do echo “$(date) | $(du -sh ~)” \
    >>/var/logs/space.txt; sleep 10; done'
  image: busybox
  name: appln
  volumeMounts:
    - name: config-volume
      mountPath: /var/logs
- image: busybox
  name: transform
  args:
  - /bin/sh
```

```
- -c
- 'sleep 30; while true; do while read LINE; do echo "$LINE" \
  | cut - f2 -d"|" >>$(date +%Y-%m-%d-%H-%M-%S)-transform.txt; \
  done < /var/logs/space.txt; sleep 20; done'
volumeMounts:
  - name: config-volume
    mountPath: /var/logs
```

The next step is to use the kubectl create command to create the multi-container Pod.

Question 9: Create an nginx Pod named check-probe. Add probes to the Pod so that if the application inside is not responding to HTTP requests on port 8080, It should be restarted.

Answer: For the solution, use YAML to create the Pod. We define the nginx Pod named check-probe, which has the HTTP method for liveness probe defined:

```
apiVersion: v1
kind: Pod
```

```
metadata:

name: check-probe



spec:

containers:

- image: nginx

name: check-probe

ports:

- containerPort: 8080

livenessProbe:

   httpGet:

     path: /

     port: 8080

   initialDelaySeconds: 5

   periodSeconds: 5
```

The next step is to use the kubectl create command to create the Pod with liveness probe.

Question 10: Create the Pod with the below YAML and identify the issue of why it is failing.

```
apiVersion: v1

kind: Pod

metadata:


  name: fail-pod

spec:

  containers:

  - args:

    - /bin/sh

    - -c

    - while true; do echo $(date) >> ~/tmp/curr-date.txt; sleep 5; done;

    image: busybox
```

name: fail-pod

Answer: The first step is to create the Pod from the YAML in the question.

Refer to Figure which shows the creation of Pod from YAML:

```
kishore@ss-MacBook-Pro chapter14 % kubectl create -f failpod.yaml
pod/fail-pod created
kishore@ss-MacBook-Pro chapter14 % kubectl get pods
NAME       READY   STATUS    RESTARTS   AGE
fail-pod   1/1     Running   0          7s
```

Figure 14.18: Pod created

The next step is to check the Pod logs for any issues:

$kubectl logs fail-pod

The above commands output the logs of the Pod.

Refer to Figure which outputs the log of the The issue is nonexistent in the directory:

```
kishore@ss-MacBook-Pro chapter14 % kubectl logs fail-pod
/bin/sh: can't create /root/tmp/curr-date.txt: nonexistent directory
/bin/sh: can't create /root/tmp/curr-date.txt: nonexistent directory
/bin/sh: can't create /root/tmp/curr-date.txt: nonexistent directory
/bin/sh: can't create /root/tmp/curr-date.txt: nonexistent directory
/bin/sh: can't create /root/tmp/curr-date.txt: nonexistent directory
/bin/sh: can't create /root/tmp/curr-date.txt: nonexistent directory
```

Figure 14.19: Pod log output

Question 11: Create three Pods that use the image nginx. The names of the Pods should be and Assign the label tier=fe to the label tier=be to and tier=db

Assign the annotation with the branch equal to the uat value to pod-1 and pod-3.

Answer: The first step is to create the three Pods with labels using the kubectl command:

$kubectl run pod1 --image=nginx -l tier=fe

$kubectl run pod2 --image=nginx -l tier=be

$kubectl run pod3 --image=nginx -l tier=db

The above command creates the Pod with corresponding labels.

The next step is to use annotate command to annotate the Pods imperatively:

$kubectl annotate pod pod1 pod3 branch="uat"

Question 12: Continuing with previous Question 11, assign All pods and pod3 the label

Use label selection to find all Pods with the team which also has label tier=be.

Answer: The first step is to assign all pods with the label

$kubectl label pod pod1 pod2 pod3 team=”avengers”

Next, Use the equality-based and set-based filter criteria to filter Pod according to the label:

$kubectl get pods -l tier=be,’team in (avengers)’

Question 13: Create a Deployment named deploy with three replicas. nginx should be the image used.

Once the deployment and Pod are in a running state image, needs to be updated to nginx:latest.

Answer: The following command creates the deployment object with three Pods with ngix image:

$kubectl create deployment my-dep --image=nginx --replicas=3

Refer to Figure which shows the deployment is created:

kishore@ss-MacBook-Pro chapter14 % kubectl create deployment my-dep --image=nginx --replicas=3
deployment.apps/my-dep created
kishore@ss-MacBook-Pro chapter14 % kubectl get deployment my-dep
NAME READY UP-TO-DATE AVAILABLE AGE
my-dep 3/3 3 3 21s

Figure 14.20: Deployment created

The next step is to update the image of the Pod:

$kubectl set image deployment/my-dep nginx=nginx:latest

The above command updates the image of the Pod:

```
kishore@ss-MacBook-Pro chapter14 % kubectl set image deployment/my-dep nginx=nginx:latest
deployment.apps/my-dep image updated
```

Figure 14.21: Update image of the Pod

Question 14: Continuing with previous Question 13, scale replicas to four.

Check the deployment rollout history. Deployment needs to be reverted to revision 1.

Answer: The first step is to use the kubectl command to scale the replicas of Pod to 4:

$kubectl scale deployments my-dep --replicas=4

The above command scales the replicas of the Pod to 4

Refer to Figure which shows the scaling of the Pod:

```
kishore@ss-MacBook-Pro chapter14 % kubectl scale deployments my-dep  --replicas=4
deployment.apps/my-dep scaled
kishore@ss-MacBook-Pro chapter14 % kubectl get deployment my-dep
NAME     READY   UP-TO-DATE   AVAILABLE   AGE
my-dep   4/4     4            4           2m40s
```

Figure 14.22: Scale the Pod replicas

The above command reverts the deployment to version 1:

$kubectl rollout undo deployment/my-dep --to-revision=1

Refer to Figure which rolls out the deployment to-revision

```
kishore@ss-MacBook-Pro chapter14 % kubectl rollout history deploy
deployment.apps/my-dep
REVISION  CHANGE-CAUSE
1         <none>
2         <none>

kishore@ss-MacBook-Pro chapter14 % kubectl rollout undo deployment/my-dep --to-revision=1
deployment.apps/my-dep rolled back
```

Figure 14.23: Revert the deployment to the previous version

Question 15: Create a cron job named date that runs every minute, executes the shell command echo “latest date: $(date)” and monitors the Job.

Answer: Create the cron job using the kubectl command:

$kubectl create cronjob date --schedule=”* * * * *” --restart=OnFailure --image=nginx -- /bin/sh -c ‘echo “latest date: $(date)”’

The above command creates the cron job with an echo command and prints the latest

Refer to Figure which shows the creation of the cron job:

```
kishore@ss-MacBook-Pro chapter14 % kubectl create cronjob date --schedule="* * * * *" --restart=OnFailure --image=nginx -- /bin/sh
-c 'echo "latest date: $(date)"'
cronjob.batch/date created
kishore@ss-MacBook-Pro chapter14 % kubectl get cronjob date
NAME   SCHEDULE    SUSPEND   ACTIVE   LAST SCHEDULE   AGE
date   * * * * *   False     0        <none>          13s
```

Figure 14.24: cron job created

Question 16: Create a deployment with image nginx with three replicas. The container needs to be exposed at port 80.

Answer: The following command creates the deployment:

$kubectl create deployment mydep --image=nginx --replicas=3 --port=80

The next step is to create the corresponding service object:

$kubectl expose deploy mydep --target-port=80

The above command exposes the service with ClusterIP and the target port

Refer to Figure which creates the deployment and corresponding service:

```
kishore@ss-MacBook-Pro chapter14 % kubectl create deployment mydep --image=nginx --replicas=3 --port=80
deployment.apps/mydep created
kishore@ss-MacBook-Pro chapter14 % kubectl expose deploy mydep --target-port=80
service/mydep exposed
kishore@ss-MacBook-Pro chapter14 % kubectl get services
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP   6d4h
mydep        ClusterIP   10.97.113.142   <none>        80/TCP    5s
```

Figure 14.25: Deployment and corresponding service created

Question 17: Create a persistent volume named ReadWriteMany is the access mode, shared is the storage class name and storage capacity is 512MB, and

/data/config1 is the host path.

Answer: The first step is to define the PV declaratively in YAML:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv1
spec:
  capacity:
    storage: 512m
  accessModes:
    - ReadWriteMany
  storageClassName: shared
  hostPath:
    path: /data/config1
```

The next step is to use the kubectl create command to create the PV.

Question 18: Continuing with previous question 17, create a Persistent Volume Claim named pvc1 that requests the Persistent Volume pv. The claim should request 256MB.

Answer: Create the PVC using the YAML and kubectl create commands:

```
kind: PersistentVolumeClaim

apiVersion: v1

metadata:

  name: pvc1

spec:

  accessModes:

    - ReadWriteMany

  resources:

    requests:

      storage: 256m
```

```
  storageClassName: shared
```

The next step is to use the kubectl create command to create the PVC.

Question 19: Continuing with previous question 18, mount the PersistentVolumeClaim pvc1 from a new Pod with image nginx and is named app1 with /var/app/config as the path.

Answer: Create a YAML file for the Pod:

```
apiVersion: v1

kind: Pod

metadata:

  name: app1

spec:

  containers:

  - image: nginx

    name: app

    volumeMounts:
```

```
    - mountPath: “/var/app/config”

      name: configpvc

  volumes:

    - name: configpvc

      persistentVolumeClaim:

        claimName: pvc1
```

The next step is to use the kubectl create command to create the Pod, which mounts the PVC

Conclusion

In this chapter, we focused on CKAD mock questions.

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Index

A

adapter pattern 83

example

ambassador pattern 82

annotations

applications

deploying, in containers 5

deploying, in Virtual Machines (VMs) 4

deployment, on physical server 2

autoscaling deployment 51

C

Certified Kubernetes Application Developer (CKAD) exam 16

environment 18

practice test core concepts 36

syllabus 17

time management 19

tips and tricks 35

CKAD exam syllabus 16

application deployment 17

application design and build 17

application environment configuration and security 17

application observability and maintenance 17

service and networking 18

Cloud Native Computing Foundation (CNCF) 16

ClusterIP 160

Cluster nodes 9

container runtime 10

Kubelet 10

kube-proxy 10

ConfigMap

creating 108

creating, from directory 112

creating, from file 110

creating, with content 111

creating, with multiple entries 109

secrets 113

Container 22

container images 6

container runtime 10

Control/Master node 8

Cron Job

D

debugging

tips and tricks 156

deployments 40

and Service 165

autoscaling deployment 51

creating 34

recreate deployment strategy

Rolling update deployment

types 42

deployments volume

configuring 94

describe command 28

describe Pod command 151

Docker 6

container images 6

containers 7

installing 12

practice test 13

registries 7

docker images command 12

docker ps command 12

docker run hello-world command 11

E

environment variables

mounting, to Pods

etcd 9

events

monitoring

F

failureThreshold 141

G

get events command 151

get pod command 32

get pods command 23

H

health probing 136

Helm

charts

practice test deployment

Helm packages 39

help command 28

I

Init container 86

scenarios 87

initialDelaySeconds 141

Input/Output (I/O) 2

J

Job

K

kube-apiserver 8

kube-controller-manager 9

kubectl cheat sheet 18

kubectl create command 34

kubectl create namespace test command 31

kubectl delete pod command 34

kubectl get cronjobs command 72

kubectl get deployment command 34

kubectl get events command 153

kubectl get namespace command 32

kubectl logs count-scqbw command 70

kubectl run command 23

Kubelet 10

kube-proxy 10

Kubernetes 8

architecture 8

benefits 11

Cluster nodes 9

etcd 9

kube-apiserver 8

kube-controller-manager 9

kube-scheduler 9

networking 166

network policy

practice test 13

Services 160

Kubernetes, benefits

declarative 11

self-healing 11

speed 10

Kubernetes cluster

debugging 146

monitoring

Kubernetes configuration 106

data, consuming 108

data, defining 108

environment variables

Kubernetes volume

types 93

kube-scheduler 9

L

labels 62

selections

liveness probe

LoadBalancer 160

logs

monitoring

M

Memory, Central Processing Unit (CPU) 2

microservices

evolution 6

minikube

installing 13

minikube delete command 13

minikube start command 13

minikube status command 13

minikube stop command 13

mock exam 1

mock exam 2

monitoring

tips and tricks 156

multi-container Pod 78

use cases 79

N

namespace 32

network policy

NodePortIP 160

O

observability

Omega 7

P

periodSeconds 141

PersistentVolumeClaim (PVC) 95

example 96

Persistent Volume (PV) 95

example 96

physical server

architecture 3

Pod 78

configuring, with PV and PVC

creating 26

declarative approach

deleting 27

environment variables, mounting

imperative approach

logs, accessing

monitoring

practice test 75

running 30

scheduling 26

volume, mounting

Pod creation command 149

Pod volume 94

configuring 93

probe 136

R

readiness probe 136

configuration 137

example 138

recreate deployment strategy

registries 7

ReplicaSet 33

resource boundary 126

resource quota

Rolling update deployment

run command 28

S

secrets 113

security context

defining

service account 130

creating 132

Services

ClusterIP 160

creating

LoadBalancer 160

NodePortIP 160

troubleshooting 174

types 160

sidecar pattern 79

authentication 80

authorization 80

examples

load balancing 80

logging 79

monitoring 79

software deployment 2

software development 2

software evolution 2

startup probe 141

example 142

storage class

using 101

successThreshold 141

T

timeoutseconds 141

troubleshooting

Services 174

V

Virtual Machines (VMs) 3

volume

mounting, to Pods

Y

Yet Another Mark-Up Language (YAML) 17